29th International Conference on Types for Proofs and Programs

TYPES 2023 – Abstracts

Eduardo Hermo Reyes and Alicia Villanueva (eds.)

Valencia (Spain), 12-15 June 2023

Foreword

This volume contains the abstracts of the talks accepted for presentation at the 29th International Conference on Types for Proofs and Programs (TYPES 2023) held in Valencia, 12-15 June 2023.

The TYPES meetings are a forum to present new and ongoing work in all aspects of type theory and its applications, especially in formalized and computer-assisted reasoning and computer programming.

The meetings from 1990 to 2008 were annual workshops of a sequence of five EU-funded networking projects. Since 2009, TYPES has been run as an independent conference series.

In response to the call for contributions, 64 abstracts were submitted from authors in 19 different countries. Two of the submissions were withdrawn by the authors. Accepted contributions were distributed in sessions covering the different topics in the call:

- foundations of type theory and constructive mathematics;
- applications of type theory;
- dependently typed programming;
- industrial uses of type theory technology;
- meta-theoretic studies of type systems;
- proof assistants and proof technology;
- automation in computer-assisted reasoning;
- links between type theory and functional programming;
- formalizing mathematics using type theory

In addition, the symposium program included invited talks by four outstanding speakers: Andrej Bauer (University of Ljubljana, Slovenia), Simona Ronchi della Rocca (Università di Torino, Italy), Marie Kerjean (LIPN, CNRS, Université Sorbonne Paris Nord, France), and Yannick Foster (Inria Nantes, France). This volume includes the abstracts of the invited talks.

Eduardo Hermo Reyes and Alicia Villanueva
Valencia, June 2023

Contents

1

Invited talks

Verified Extraction from Coq to OCaml

Yannick Foster

Inria Nantes, France

Abstract

A central claim to fame of the Coq proof assistant is extraction to languages like OCaml, centrally used in landmark projects such as CompCert. Extraction was initially conceived and implemented by Pierre Letouzey, and is still guiding design decisions of Coq's type theory. While the core extraction algorithm is verified on paper, central features like optimisations –of which there are 10 the user can enable– only have empirical correctness guarantees.

In the scope of the MetaCoq project, which aims at placing Coq's type theory on a well-defined and fully formal foundation, I am working with other members of the MetaCoq team on a re-implementation and verification of all aspects of Coq's extraction process to OCaml. The new extraction process is based on a formal semantics of Coq as provided by MetaCoq and a formal semantics of the intermediate language of the OCaml compiler derived from the Malfunction project.

In my talk, I plan on discussing the current state of this verification, its goals, possible extensions, and design decisions along the way, a discussion of the trusted computing and theory bases (and in particular ideas for reducing them), arising problems with Coq and the surrounding infrastructure, and the impact on other projects. I will conclude with thoughts on how other proof assistants can learn andbenefit from the lessons we have learned.

The differentiation monad

Marie Kerjean (invited speaker)[1] and Jean-Simon Pacaud Lemay[2]

[1] CNRS, LIPN
Université Sorbonne Paris Nord
kerjean@lipn.fr
[2] School of Mathematical and Physical Sciences
Macquarie University
Email: js.lemay@mq.edu.au

This talk is based on joint work with Jean-Simon Pacaud Lemay [KL].

The continuation monad is a very basic programming structure, at the heart of several other ones. Its unit very basically embeds a value into its continuation:

$$v \mapsto \lambda k.kv : A \Rightarrow (A \Rightarrow B) \Rightarrow B$$

What happens when we enforce the linearity of certain arrows above ? Linearity here is meant in the Linear Logic [Gir87] sense: linear implication $\multimap$ are interpreted as linear maps in appropriate algebraic models, or corresponds to proofs that use exactly once their hypothesis.

$$v \mapsto \lambda k.D[k]v : A \multimap (A \Rightarrow B) \multimap B$$

While the rightmost $\multimap$ is linear by construction, making leftmost $\multimap$ linear means that the continuation k is now a linear map: we differentiated it.

This operation in fact already existed as an inference rule in Differential Linear Logic (DiLL) [ER06]. Differential Linear Logic is an extremely symmetrical extension of Linear Logic. When the second allows to handle the fact that linear proofs are in particular non-linear, the second allows to do the converse transformation, going from non-linear proofs to linear proofs, that is differentiating them. It operates on the type $!A := (A \Rightarrow \bot) \multimap \bot$ of distributions with compact support.

$$\bar{d} := v \mapsto (f \mapsto D_0(f)) : A \multimap\, !A$$

The connective ! is the fundamental introduction of Linear Logic, interpreted as the space of distributions in models of classical Linear Logic. It is historically modeled as a co-monad $(!, d, p)$ in its denotational models. The co-unit d corresponds to the dereliction of functions as non-linear functions, and p allows non-linear function to compose. The fundamental call-by-name translation of Intuitionistic Logic into Linear Logic is the exact translation of the symmetry at the heart of distribution theory, saying that functions act on value exactly as distributions act on function.

$$A \Rightarrow B \equiv\, !A \multimap B$$

In this talk we will explain how to make ! a monad, thanks to differentiation and the convolutional exponential.

For ! to be a monad with $\bar{d}$ as a unit, we are looking for a multiplication law $\bar{p} : !!A \multimap\, !A \equiv\, !A \Rightarrow\, !A$. The monads law will tell us in particular that $\bar{d}; \bar{p} = id$, which means that when we consider $\bar{p}$ as a non-linear map its differential at 0 must be the identity. To make $\bar{p}$ coherent with Differential Linear Logic, we must have it as a monoid morphism on !, meaning that $\bar{p}$ applied to a convolution of distributions results in the multiplication of the actions of $\bar{p}$. A morphism whose differential at 0 is the identity, and who transforms sums into multiplication is nothing but the convolutional exponential:

$$\bar{p} := \phi \mapsto \sum_n \frac{1}{n!}\phi^{*^n} : !A \Rightarrow !A \equiv !!A \multimap A$$

We will dive into the properties of the differentiation monad $(!, \bar{d}, \bar{p})$. In particular, we will show that while it is known that the interaction $\bar{d}$ and p is the exact translation of the chain rule in analysis, the interaction between $\bar{p}$ and d is exactly a symmetric "co-chain" rule. Most importantly, we will show that the monad law $!\bar{d}; \bar{p} = id$ is verified only if we place ourselve into a *quantitative model* of lambda-calculus.

The quantitative point of view on programming languages consists in measuring through syntax, types or models their usage in time or resources. This has in particular led to refined results for the λ-calculus and innovations in probabilistic programming. In denotational semantics, it typically consists of interpreting programs by power series, whose coefficients represent the quantitative information one would like to retrieve. In an analytic context, power series are in particular functions which equal their Taylor series at 0:

$$f(v) = \sum_{n \in \mathbb{N}} \frac{1}{n!} D_0^{(n)}(f)(v).$$

As $(D_0(\_)(v))^{*^n} = D_0^{(n)}(\_)(v)$, the above equation is the exact translation of $!\bar{d}; \bar{p} = id$ on every function $f : A \Rightarrow \perp$ and vector $v : A$.

However, finding concrete interpretation of the monad $(!, \bar{d}, \bar{p})$ is not so easy. Indeed, p and $\bar{p}$ do not interact very well: for $\phi : !A$, $\bar{p}(\phi)$ as a sum of distributions might converge on functions f g, but not on their composition $f \circ g$. The solution comes from independent studies on the convolutional exponential in functional analysis [GHOR00]. We will show that this gives us a graded monad, reconciling Differential Linear Logic with Graded Linear Logics.

References

[ER06] T. Ehrhard and L. Regnier. Differential interaction nets. *Theoretical Computer Science*, 364(2), 2006.

[GHOR00] R. Gannoun, R. Hachaichi, H. Ouerdiane, and A. Rezgui. Un théorème de dualité entre espaces de fonctions holomorphes à croissance exponentielle. *Journal of Functional Analysis*, 171(1), 2000.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.

[KL] Marie Kerjean and Jean-Simon Pacaud Lemay. Taylor Expansion as a Monad in Models of DiLL. In *LICS 2023*.

Intersection and Simple types

Simona Ronchi della Rocca

Università di Torino, Italy

Abstract

When, in the seventies of the last century, Coppo and Dezani designed intersection types, their main motivation was to extend the typability power of simple types, adding them an intersection connective, enjoying associativity, commutativity and idempotency, so denoting set formation. In fact, the simple types system can be seen as a restriction of the intersection type system where all sets are singletons. Quite early intersection types turned out to be useful in characterizing qualitative properties of λ-calculus, like solvability and strong normalization, and in describing models of λ-calculus in various settings. A variant of intersection types, where the intersection is no more idempotent, has been recently used to explore quantitative properties of programming language, like the length of the normalisation procedure. It is natural to ask if there is a quantitative version of the simple type system, or more precisely a decidable restriction of non-idempotent intersection system with the same typability power of simple types. Since the lack of idempotency, now the intersection corresponds to multiset formation, so (extending the previous reasoning) the natural answer is to restrict the multiset formation to copies of the same type. But this answer is false, the so obtained system is decidable, but it has less typability power than simple types. We prove that the desired system is obtained by restricting the multiset formation to equivalent types, where the equivalence is an extension of the identity, modulo the cardinality of multisets.

On Isomorphism Invariance and Isomorphism Reflection in Type Theory

Andrej Bauer
University of Ljubljana

Isomorphism Invariance, also called the Principle of Isomorphism and the Principle of Structuralism, is the idea that isomorphic mathematical objects share the same structure and relevant properties, or that whatever can reasonably be done with an object can also be done with an isomorphic copy:

Isomorphism Invariance: "If $A \cong B$ and $\Phi(A)$ then $\Phi(B)$."

Without any limitation on Φ, we may take $\Phi(X)$ to be $A = X$ and obtain:

Isomorphism Reflection: "If $A \cong B$ then $A = B$."

Conversely, Isomorphism Reflection implies Isomorphism Invariance by Leibniz's Identity of Indiscernibles. Depending on how we interpret $\cong$ and $=$, these principles might be plainly false, uninspiringly true, or a foundational tenet:

1. In set theory, if $\cong$ is existence of a bijection, the principles are false.
2. In set theory, if $\cong$ is isomorphism of sets qua $\in$-structures, both principles are true because Isomorphism Reflection is just the Extensionality axiom.
3. In type theory, if $=$ is propositional equality and $\cong$ is equivalence of types, Isomorphism Reflection is (a consequence of) the Univalence axiom.

While Isomorphism Invariance is widely used in informal practice, with $\cong$ understood as existence of structure-preserving isomorphism, Isomorphism Reflection seems quite unreasonable because it implies bizarre statements, e.g., that there is just one set of size one. Any formal justification of the former must therefore address the tension with the latter principle.

In this talk I will review what is known about the formal treatment of the principles, recall the cardinal model of type theory by Théo Winterhalter and myself which shows that Isomorphism reflection is consistent when $=$ is taken as judgemental equality, and discuss the possibility of having other models validating judgemental Isomorphism reflection that might be compatible with non-classical reasoning principles. I shall also touch on the possibility of a restricted form of Isomorphism reflection that would provide a satisfactory formal definition of "canonical isomorphism".

2

Session 3: Proof assistants and proof technology

Decidable Type-Checking for Bidirectional Martin-Löf Type Theory

Meven Lennon-Bertrand and Neel Krishnaswami

University of Cambridge, UK

Abstract

In this work, we describe a presentation of dependent type systems rooted in bidirectional ideas, carefully separating at the syntax level between inferring and checking terms. This leads to a system which requires exactly the annotations needed to decide type-checking. Moreover, it readily embeds the two most usual ways to handle typing in dependent type systems, either by restricting to a certain subclass of terms that do not need annotations (as done in AGDA), or by demanding certain annotations to be provided (as done in COQ), providing an elegant unifying framework.

1 Dependent type-checking is more subtle than you think

While a large body of work has been devoted to showing decidability of conversion for various complex dependent type systems, decidability of typing has attracted comparatively little interest. However, it is a more subtle question than on can at first think.

Type-checking for dependent type systems is, in general, undecidable [3]. The main culprits are (β-)redexes: when considering $f\ u$, a type-checker typically infers a type $\Pi x\colon A.B$ for the function f, then checks u against A. However, when the function f is an abstraction $\lambda x.t$, there is nowhere this A can be obtained from! Trying to infer A from u runs into a similar issue.

There are two standard approaches to get out of this difficulty. The first one is to restrict the terms fed to the type-checker to a subset for which type-checking becomes decidable again. The kernel of the AGDA proof assistant, for instance, only manipulates terms in normal forms [9], for which type-checking is generally decidable whenever conversion is [1, 5].

The second approach is to decorate terms with type annotations, to ensure one can always *infer* a type for any typable term. For instance, in COQ or LEAN, the kernel only deals with annotated abstractions $\lambda x\colon A.t$, which contain exactly the information we were missing in the redex earlier. Again, this leads to decidable type-checking when conversion is decidable [6].

In practice, these limitations are mitigated at the user level using unification, allowing for a syntax which is more flexible than the kernel's. Still both approaches still have significant drawbacks. Because AGDA can only represent normal forms, it has to eagerly normalize terms, and cannot type-check intermediate steps of its reduction machine. Because COQ can only represent terms that infer, it can be quite inefficient, as annotations create a lot of redundancy. Finally, elaboration based on unification is inherently incomplete. This makes them less predictable, forcing users to get an intuition as to which unification problems their tool can solve. Designing a system which has a complete, and thus predictable, type-checking, and has less of the aforementioned shortcomings, thus seems like a desirable goal.

2 A bidirectional analysis

To better understand the issue, we can turn to bidirectional typing [4], an analysis of type-checking algorithms which emphasizes the difference between two *modes*, between which most such algorithms alternate: checking–where the type is known–and inference–where it is to be found. In this view, the issue with our redex is that in an application $f\ u$ we want the function f to infer a type, but an abstraction $\lambda x.t$ can only be reasonably checked. Both solutions sketched in Section 1 are quite radical: Agda forces f to be a neutral term which always infers, while Coq demands that *all* terms infer a type. What if, instead, we simply demanded that f inferred a type, whatever way it achieves this? What if we separated, already in the syntax, between inferring terms and checking terms to be able to express this demand?

This idea has already appeared in the literature [5, 8]. Both works, however, use it only to relax Agda-style type-checking, by adding an annotation $t :: A$ to a language otherwise completely devoid of them, allowing writing a β-redex as $(\lambda x.t :: \Pi x\colon A.B)\ u$. But this is heavier than Coq-style annotations, because in general only the domain annotation is really needed. So, let us simply throw these in as well! Altogether, the functional fragment of such a language looks as follows, divided in the two, mutually defined checking and inferring terms:

$$c ::= \underline{i} \mid \lambda x.c \qquad\qquad i ::= c :: A \mid x \mid i\ c \mid \lambda x\colon A.i \mid \Pi x\colon A.B \mid \square_k$$

with both typed and untyped abstractions, types (in the inferring fragment, as we luckily can infer their type), and $\underline{i}$, the converse of annotation that forgets that its argument is inferring. While not presented here for lack of space, this presentation easily extends to virtually any feature present in modern dependent type systems, such as inductive types or negative records.

Because we design the language to respect the structure of bidirectional algorithms, these work perfectly, and typing is decidable for systems in this fashion as soon as conversion is. Moreover, both earlier approaches can be carved out as subsystems: the first, by using neither $t :: A$ nor annotated abstraction; the second, by restricting to the inferring fragment, using no checking term but $\underline{i}$. We thus get two proofs of decidability for the price of one.

3 Substitution, reduction and conversion

Of course, this system would not be any good without a well-behaved conversion. Before coming to it, however, we must beware of substitution. The naïve reduction of $(\lambda x\colon A.t)\ u$ to $t[x := u]$ is incorrect, as it does not preserve modes: the variable x infers, but u merely checks, and so replacing one by the other is incorrect. Rather, the correct β-redex is instead $t[x := u :: A]$, as it is *mode-preserving*. If we care about preservation of typability during a small step reduction chain, typically to be able to read back terms to users, then this is the correct substitution rule.

The second point of interest are of course annotations. Here we can take inspiration from the transport of observational equality [2, 10] or the casts of gradual typing [7]: annotations reduce when both the type and terms are canonical, and propagate down: $(\lambda x.t) :: (\Pi x\colon A.B) \to \lambda x\colon A.(t :: B)$. Combining this with the previous rule for β-redex, we recover the earlier rule of McBride [8]. Conversely, we can erase useless annotations: $\underline{\lambda x\colon A.t} \to \lambda x.\underline{t}$. Finally, we also want some form of extensionality for annotations: $\underline{t :: A}$ should be convertible to t.[1] Just as for casts on reflexivity in the latest version of observational equality [11], however, we do not want to see this as a reduction rule, but rather as an equation only needed on neutral terms. In the end, we get a system close to that of Pujet and Tabareau [11], and we expect being able to adapt their technique to show decidability of our conversion.

If, however, we are only interested in evaluation to decide conversion, we know that normal forms do not contain annotations. Why, then, bother with maintaining those? Instead, we can erase them in a normalization by evaluation approach, similar to e.g. Gratzer et al. [5]. This speeds up conversion-checking, at the cost of losing a well-typed reduction sequence.

[1] Note that it does not make sense to compare simply $t :: A$ and t, as they do not have the same mode.

References

[1] Andreas Abel and Gabriel Scherer. "On Irrelevance and Algorithmic Equality in Predicative Type Theory". In: *Logical Methods in Computer Science* Volume 8, Issue 1 (Mar. 2012). DOI: 10.2168/LMCS-8(1:29)2012. URL: https://lmcs.episciences.org/1045.

[2] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "Observational Equality, Now!" In: *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*. PLPV '07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 57–68. ISBN: 9781595936776. DOI: 10.1145/1292597.1292608.

[3] Gilles Dowek. "The undecidability of typability in the Lambda-Pi-calculus". In: *Typed Lambda Calculi and Applications*. Ed. by Marc Bezem and Jan Friso Groote. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 139–145.

[4] Jana Dunfield and Neel Krishnaswami. "Bidirectional Typing". In: *ACM Computing Surveys* 54.5 (May 2021). ISSN: 0360-0300. DOI: 10.1145/3450952.

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

[6] Meven Lennon-Bertrand. "Bidirectional Typing for the Calculus of Inductive Constructions". PhD thesis. Nantes Université, 2022.

[7] Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. "Gradualizing the Calculus of Inductive Constructions". In: *ACM Transactions on Programming Languages and Systems* 44.2 (Apr. 2022). ISSN: 0164-0925. DOI: 10.1145/3495528.

[8] Conor McBride. "Types Who Say Ni". 2022.

[9] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007.

[10] Loïc Pujet. "Computing with Extensionality Principles in Type Theory". PhD thesis. Nantes Université, 2022.

[11] Loïc Pujet and Nicolas Tabareau. "Impredicative Observational Equality". In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023). DOI: 10.1145/3571739.

Building an elaborator using extensible constraints

Bohdan Liesnikov[1], Jesper Cockx[1]

TU Delft, Delft, Netherlands

Dependently-typed languages proved to be very useful for statically enforcing properties of programs and for enabling type-driven development. However their implementations have been studied to a smaller extent than their theoretical properties. Theoretical models of these languages do not consider the plethora of features that exist in a bigger language like Agda, leading to issues in the implementation of the unifier and the elaborator. We present work-in-progress on a new design for elaboration of dependently-typed languages based on the idea of an open datatype for constraints to tackle these issues. This allows for a more compact base elaborator implementation while enabling extensions to the type system. We do not require modifications to the core of type-checker, therefore preserving safety of the language.

Introduction. The usual design of a compiler for a dependently-typed language consist of four main parts: a parser, an elaborator, a core type-checker, and a back-end. Some languages omit some parts, such as Agda which lacks a full core type-checker. The elaborator can further be divided into two parts: traversal of the terms with collection of the constraints and solving of the constraints [8]. These can be found in all major dependently-typed languages like Idris, Coq, Lean, and Agda, though they are at times interleaved. Agda perhaps pushes the idea of constraints the furthest and uses a family of 17 kinds of constraints. We will focus on it specifically below since the problems are most prominent there.

Problems with unifiers. The most common constraint type is equality, which is typically solved by a unifier. In order to provide the most powerful inference to users, compiler writers often extend the unifier to make it more powerful, which leads to complex and intricate code. This code is also heavily used throughout the compiler: either as direct functions `leqType` when type-checking terms, `compareType` when type-checking applications, or as raised constraints `ValueCmp` and `SortCmp` from `equalTerm` while checking applications or definitions, `ValueCmpOnFace` from `equalTermOnFace` again while checking applications. This makes it sensitive towards changes and hard to maintain and debug.

An example from Agda's conversion checker is the `compareAs` function which provides type-directed unification and yet the vast majority of it are special cases for metavariables. This function calls the `compareTerm'` function which then calls the `compareAtom` function. Each of the above functions implements part of the "business logic" of the conversion checker with the total line count above 400 lines. But each of them contains a lot of code dealing with bookkeeping related to metavariables and constraints: they have to throw and catch exceptions, driving the control flow of the unification, compute blocking tags that determine when a postponed constraint is retried, and deal with cases where either or both of the sides equation or its type are either metavariables or the reduction is blocked on one. As a result this code is unintuitive and full of intricacies as indicated by multiple comments in the source code.

Zooming in on the `compareAtom` function, the actual logic can be expressed in about 20 lines of simplified code. This is precisely what we would like the compiler developer to write, not to worry about the dance around the constraint system.

The functions described above are specific to Agda but in other major languages we can find similar problems with unifiers being large modules that are hard to understand. The sizes of modules with unifiers are as follows: Idris (1.5kloc), Lean (1.8kloc), Coq (1.8kloc). For Haskell, which is not a dependently-typed language yet but does have a constraints system [7], this number is at 2kloc.

How do we solve this? While Agda relies on constraints heavily, the design at large does not put them in the centre of the picture and instead frames them as a gadget. To give a concrete example, functions `noConstraints` or `dontAssignMetas` rely on specific behaviour of the constraint solver system and are used throughout the codebase. `abortIfBlocked`, `reduce` and `catchConstraint`/`patternViolation` force the programmer to make a choice between letting the constraint system handle blockers or doing it manually. These things are known to be brittle and pose an increased mental overhead when writing a type-checker.

Our idea for a new design is to shift focus more towards the constraints themselves: First we give a stable API for raising constraints that can be called by the type-checker, essentially creating an "ask" to be fulfilled by the solvers. This is not dissimilar to the idea of mapping object-language unification variables to host-language ones as done by Guidi, Coen, and Tassi [5], view of the "asks" as a general effect [3, ch. 4.4] or communication between actors [1]. Second, to make the language more modular we make constraints an extensible data type in the style of Swierstra [9] and give an API to define new solvers with the ability to specify what kinds of constraints they can solve. Our prototype is implemented in Haskell as is available at github.com/liesnikov/extensible-elaborator.

For example, to solve unification problems we need to define a constraint that models them:

```
data EqualityC e = EqualityCC Term Term Type
```

On the solver side we need to define a suite of unification solvers that handle different cases of the problem. Let us take a look at the simplest example – checking syntactic equality.

```
syntacticH :: (MonadElab m, EqualityC :<: c)=> Constraint c -> m Bool
syntacticS :: (MonadElab m, EqualityC :<: c)=> Constraint c -> m ()
syntactic = Plugin {handler=syntacticH, solver=syntacticS,
                    pre=[...], suc=[...], tag="syntactic"}
```

We first define the class of constraints that will be handled by the solver by providing a "handler" – a function that decides whether a given solver has to fire. In this case – checking that the constraint given is indeed an `EqualityC` and that the two terms given to it are syntactically equal. The solver in this case simply marks the constraint as solved, since it only fires once it has been cleared to do so by a handler. We separate the handler from the solver to allow for cheaper decision procedures and more expensive, effectful solvers. Finally, we register the solver by declaring it using a plugin interface specifying solvers that precede and succeed it.

Open constraint datatype. Refactoring the unifier into smaller solvers results in a compact elaborator for a simple language. Moreover, making the constraint datatype open and allowing users to register new solvers allows us to extend the language without affecting the core. For example, to add implicit arguments to the language it is enough to extend the parser, add one case to the elaborator to add a new meta for every implicit and register a solver. For a simple implicit every such metavariable will be instantiated by the unifier. Once we have implicits as a case in the elaborator we believe that the design can accommodate type classes [6] and tactic arguments [10, ch. 3.17.1] with just additional solvers and parsing rules. We hope to also implement coercive subtyping (akin to [2]) and, perhaps, row types [4].

References

[1] Guillaume Allais et al. "TypOS: An Operating System for Typechecking Actors". In: TYPES. Nantes, France, June 22, 2022, p. 3. URL: https://types22.inria.fr/files/2022/06/TYPES_2022_paper_31.pdf.

[2] Andrea Asperti et al. "Crafting a Proof Assistant". In: *Types for Proofs and Programs*. Ed. by Thorsten Altenkirch and Conor McBride. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2007, pp. 18–32. ISBN: 978-3-540-74464-1. DOI: 10.1007/978-3-540-74464-1_2.

[3] Andrej Bauer, Philipp G. Haselwarter, and Anja Petković. "Equality Checking for General Type Theories in Andromeda 2". In: *Mathematical Software – ICMS 2020*. Ed. by Anna Maria Bigatti et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 253–259. ISBN: 978-3-030-52200-1. DOI: 10.1007/978-3-030-52200-1_25.

[4] Benedict R Gaster and Mark P Jones. *A Polymorphic Type System for Extensible Records and Variants*. Technical Report NOTTCS-TR-96-3. Department of Computer Science, University of Nottingham, 1996.

[5] Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. *Implementing Type Theory in Higher Order Constraint Logic Programming*. Nov. 17, 2017. URL: https://hal.inria.fr/hal-01410567 (visited on 2021-03-24).

[6] Cordelia V. Hall et al. "Type Classes in Haskell". In: *ACM Transactions on Programming Languages and Systems* 18.2 (Mar. 1, 1996), pp. 109–138. ISSN: 0164-0925. DOI: 10.1145/227699.227700.

[7] Simon Peyton Jones. "Type Inference as Constraint Solving: How GHC's Type Inference Engine Actually Works". June 15, 2019. URL: https://www.microsoft.com/en-us/research/publication/type-inference-as-constraint-solving-how-ghcs-type-inference-engine-actually-works/ (visited on 2022-06-28).

[8] N.G. de Bruijn. "A Plea for Weaker Frameworks". In: *Logical Frameworks*. Ed. by G. Huet and G. Plotkin. Cambridge: Cambridge University Press, 1991, pp. 40–67. ISBN: 978-0-521-41300-8.

[9] Wouter Swierstra. "Data Types à La Carte". In: *Journal of Functional Programming* 18.4 (July 2008), pp. 423–436. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796808006758.

[10] The Agda Team. *Agda User Manual*. Jan. 30, 2023. URL: https://agda.readthedocs.io/_/downloads/en/v2.6.3/pdf/ (visited on 2023-05-22).

From Lost to the River: Embracing Sort Proliferation

Gaëtan Gilbert, Pierre-Marie Pédrot, Matthieu Sozeau, and Nicolas Tabareau

Inria Rennes – Bretagne Atlantique, LS2N

Since their inception, proof assistants based on dependent type theory have featured some way to quantify over types. Leveraging dependent products, the most common way to do so is to introduce a type of types, known as a *universe*. Care has to be taken, as paradoxes lurk in the dark. Martin-Löf famously introduced in his seminal type theory MLTT a universe $\mathcal{U}$ with the typing rule $\mathcal{U} : \mathcal{U}$, only for Girard to show that this system was inconsistent. The standard solution is to introduce a hierarchy of universes $(\mathcal{U}_i)_{i\in\mathbb{N}}$ and mandate that $\mathcal{U}_i : \mathcal{U}_{i+1}$.

While trivial from the point of view of the typing rules, this additional index is a major source of non-modularity. One has indeed to pick a level in advance for every universe instance they write, leading to potential conflicts later on. Historically, the first answer to this problem was the introduction of floating universes, i.e. replacing $\mathbb{N}$ with a well-founded graph and checking constraints on-the-fly. This solution is simple and works for most practical uses, but is too limited for in-depth universe manipulation as it still forces a global assignment of levels.

Properly solving the issue requires a bit more expressivity, provided by universe polymorphism. Several variants of such a mechanism exist, which can roughly be put on a spectrum of internalization, from McBride's crude but effective stratification [3], to Agda where universe levels are inhabitants of a *bona fide* type [6]. On the midpoint sit the type theories of Coq [5] and Lean [2] which only allow an external, prenex form of universe polymorphism. These systems are a sweet spot as they are conservative while restoring the lost modularity. So, is everything perfect in the best of all universes?

Unfortunately for the user, but fortunately for us, the answer is no. Universe polymorphism is optimal only when there is a single hierarchy of universes. In proof assistants based on CIC, like Coq or Lean, the universe structure follows the PTS tradition, insofar as it has not one single hierarchy, but actually two. Namely, while there is on the one hand the $\texttt{Type}_i$ hierarchy that corresponds to $\mathcal{U}_i$ from MLTT, there is also a universe of propositions `Prop`[1]. The `Prop` universe is a hodgepodge of several features, mixing impredicativity, compatibility with proof-irrelevance and erasability. In order to make this sound, inductive types living in `Prop` cannot be eliminated in general into `Type`. This is a source of non-monotonicity, and as a result `Prop` cannot be treated as a level in universe polymorphism. Not only this forces code duplication between `Type` and `Prop`, but this has also annoying consequences in unification where some sorts must be explicitly annotated.

The situation in Coq has recently gotten even worse with the introduction of a third kind of universe, `SProp`, which classifies definitionally irrelevant types. Thus we have to triplicate all our definitions, but that is the least of our problems. Conversion now depends on the knowledge that a type lives in `SProp`, which in the Coq case prevents the cumulativity relation $\texttt{SProp} \subseteq \texttt{Type}$. This introduces even more code duplication compared to `Prop`. Worse, this completely breaks unification. Up to Coq 8.17, unification picked the sort of a type eagerly to be `Type`. This worked with a few quirks for $\texttt{Prop} \subseteq \texttt{Type}$, but this prevented conversion from relying on irrelevance without explicit `SProp` annotations. The kernel also had to perform a hackish "repair" of ill-annotated terms produced by elaboration.

One could claim that the existence of three hierarchies is an annoyance. We claim that in reality one should desire *many more* hierarchies. The literature abounds in examples, e.g. the

[1] We voluntarily ignore the case of impredicative `Set`.

opposition between strict and fibrant types of 2LTT [1] and the one between pure and effectful types [4]. A decent proof assistant should thus make it possible to write modular code not only w.r.t. universe levels, but also w.r.t. universe hierarchies! As a solution, we propose a novel mechanism of *sort polymorphism* complementary to the *universe polymorphism* mechanism.

In a nutshell, it introduces a new algebra of sorts s, which in the case of Coq is

$$s ::= \alpha \mid \texttt{Type} \mid \texttt{Prop} \mid \texttt{SProp}$$

where α ranges over sort variables. Sorts are then factorized as a single term constructor $\texttt{Sort}_i^s$ where s is a sort and i is a level. The usual sorts are then defined as e.g. $\texttt{Type}_i := \texttt{Sort}_i^{\texttt{Type}}$ and $\texttt{Prop} := \texttt{Sort}_0^{\texttt{Prop}}$. Just like levels, we now allow prenex quantifications over sort variables.

For this to work properly, we need the typing rules to be stable by sort instantiation. For the negative fragment, we expect the sort of a sort to be Type and the sort of a product to be the sort of its codomain, as per the rules below.

$$\frac{}{\texttt{Sort}_i^s : \texttt{Type}_{i+1}} \qquad \frac{\vdash A : \texttt{Sort}_i^{s'} \qquad x : A \vdash B : \texttt{Sort}_j^s}{\vdash \Pi(x : A).\, B : \texttt{Sort}_{i \vee j}^s}$$

This allows transparently handling impredicative universes, setting e.g. $\texttt{Sort}_i^{\texttt{Prop}} \equiv \texttt{Sort}_j^{\texttt{Prop}}$ for all levels i, j. More generally, these rules seems to be valid for every instance from the literature.

We are currently working on porting this system from the negative fragment to the more complex case of inductive types. The avowed goal is to emulate the infamous template polymorphism feature of Coq, which is a primitive form of level polymorphism with a bit of sort polymorphism blended in. Notably, this allows Coq to type $A \times B : \texttt{Prop}$ whenever $A, B : \texttt{Prop}$. We envision a system leveraging the difference between squashed types, with identity eliminations for sort variables and explicit elimination rules for ground sorts, and unsquashed types, allowing all eliminations.

This mechanism has been partially implemented in the unification algorithm of Coq 8.18. The kernel does not feature a way to quantify over sorts at the level of definitions yet, but the elaboration algorithm now manipulates algebraic sorts. In particular, relevance annotations are parameterized by sort variables, solving the aforementioned issues with SProp. As a byproduct longstanding issues due to the eager choice of Type vs. Prop were solved.

References

[1] D. Annenkov, P. Capriotti, and N. Kraus. Two-level type theory and applications. *CoRR*, abs/1705.03307, 2017.

[2] M. Carneiro. The type theory of Lean. Master's thesis, Carnegie-Mellon University, 2019.

[3] C. McBride. Crude but effective stratification, 2002.

[4] P. Pédrot, N. Tabareau, H. J. Fehrmann, and É. Tanter. A reasonably exceptional type theory. *Proc. ACM Program. Lang.*, 3(ICFP), 2019.

[5] M. Sozeau and N. Tabareau. Universe Polymorphism in Coq. In G. Klein and R. Gamboa, editors, *ITP 2014*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2014.

[6] The Agda team. The Agda manual, 2022. https://agda.readthedocs.io/.

3

Session 4: Foundations of type theory and constructive mathematics

Lax-Idempotent 2-Monads, Degrees of Relatedness, and Multilevel Type Theory

Andreas Nuyts

imec-DistriNet, KU Leuven, Belgium

Parametricity and Degrees of Relatedness Fourty years ago, Reynolds [Rey83] formulated his model of relational parametricity for (predicative [Lei91, Rey84]) System F. This was later reorganized as a model of System Fω and dependent type theory in reflexive graphs [Atk12, AGJ14], which evolved further into a cubical model [BCM15, NVD17] in order to support parametricity w.r.t. proof-relevant relations, as well as internal iterated parametricity.

Most accounts of parametricity for dependent type theory do not satisfy Reynolds' identity extension lemma – a.k.a. discreteness – for large types [AGJ14, BCM15, CH20]. The lemma and the discreteness condition express that homogeneous (i.e. non-dependent) graph edges are reflexive (i.e. constant), so that the edge relation in general can be understood as heterogeneous equality. The fact that identity extension can be satisfied for small types [AGJ14] actually has little to do with size; the reason is simply that discrete types are closed under all the usual type formers, except for the universe, which can be excluded by requiring smallness. This coupling between universe level (a device for safeguarding predicativity) and discreteness breaks down upon introducing HITs [Uni13, ch. 6] with edge constructors, or a discrete truncation, or certain modal types. Therefore, it is better to introduce an orthogonal stratification. In the type system RelDTT for Degrees of Relatedness [ND18], universes are annotated by a level as well as a *depth* $p \geq -1$ indicating the relational complexity of the types they classify. The idea is that a type of depth p is equipped with $p+1$ proof-relevant reflexive relations (as well as equality and 'true')

$$x = y \quad \Rightarrow \qquad x \frown_0 y \quad \Rightarrow \quad x \frown_1 y \quad \Rightarrow \quad \dots \quad \Rightarrow \quad x \frown_p y \qquad \Rightarrow \quad \top,$$

where discrete types must satisfy the identity extension lemma w.r.t. $\frown_0$, making $\frown_0$ a notion of heterogeneous equality. Data types such as Bool and Nat have depth 0, whereas the (discrete) universe of depth p types has depth $p+1$. RelDTT can in fact be seen as an instance of multimode type theory (MTT) [GKNB21] instantiated on a specific *mode theory* DoR [Nuy20, §9.3]. The objects of the 2-category DoR are the depths $p \geq -2$, which serve as the modes of the theory. Modalities $\mu : p \to q$ are specified by monotone functions $- \cdot \mu : \{0 \leq \ldots \leq q\} \to \{= \leq 0 \leq \ldots \leq p \leq \top\}$, denoted as $\langle 0 \cdot \mu, \ldots, q \cdot \mu\rangle$.[1][2][3] The modal type $\langle \mu \mid A\rangle$ is then conceptually the same type as A, but the ith relation of $\langle \mu \mid A\rangle$ is the $(i \cdot \mu)$th relation of A. Modal functions are (at least semantically) functions whose domain is a modal type, so that $e : x \frown_{i \cdot \mu} y$ is sent to $f(e) : f(x) \frown_i f(y)$. The identity function is then continuous ($\mathbf{con} : p \to p$ with $i \cdot \mathbf{con} = i$), while polymorphic functions may be parametric ($\mathbf{par} : p+1 \to p$ with $i \cdot \mathbf{par} = i+1$) or ad hoc ($\mathbf{hoc} : q \to p$ with $i \cdot \mathbf{hoc} = (=)$). Algebras depend on their structure via the structural modality $\mathbf{str} : p \to p+1$ such that $\mathbf{par} \circ \mathbf{str} = \mathbf{con}$.

Depth p types were originally modelled in presheaves over the category DCube_p of *depth p cubes*, which is the free cartesian-category-with-terminal-object-$\top$ over the diagram

$$\begin{tikzcd} \top \arrow[r, bend left] \arrow[r, bend right] & (\!|0|\!) \arrow[r] & \dots \arrow[r] & (\!|p|\!). \end{tikzcd}$$

This specialized model proved soundness of RelDTT, but the fact that it was constructed specifically for this occasion was at odds with the claim that RelDTT and its semantics explain the existence and behaviour of many modalities found implicitly or explicitly throughout the literature.

[1]The domain becomes empty if $q = -1$. By convention, the depth -2 is a freely added strict initial object.

[2]By a combinatorial argument, DoR is isomorphic to the semisimplex 2-category.

[3]Actually, RelDTT internalizes only the morphisms in DoR that have a left adjoint in DoR (and does not use depth -2), which for the original model was an unnecessary restriction inspired by the perceived need for a left division operation respecting context extension. For the general model below, we need to apply the same restriction so as to ensure the existence of a left adjoint operation on contexts for every internal modality.

Multilevel Type Theory Two-level type theories (2LTT) [Voe13, ACK16, ACKS17] are type systems built on top of another type system (called the *inner* system; in any treatment that I am aware of this is immediately specialized to HoTT) by internalizing aspects of its metatheory (such as extensional equality in the HoTT application) which can then be reasoned about in the *outer* system. Annenkov et al. give a general model for 2LTT: if the inner system is modelled in a category $\mathcal{C}$ (potentially its category of syntactic contexts and substitutions) then the outer system can be modelled in the presheaf category $\mathrm{Psh}(\mathcal{C})$, which contains $\mathcal{C}$ via the Yoneda-embedding. We can of course iterate this idea, which we call multilevel type theory: viewing the outer system as the inner one, we can add a further system modelled in $\mathrm{Psh}(\mathrm{Psh}(\mathcal{C}))$. This construction exhibits properties reminiscent of RelDTT. Given two objects $c, d \in \mathrm{Obj}(\mathcal{C})$, we can either first embed them in $\mathrm{Psh}(\mathcal{C})$ and then take the coproduct, yielding $\mathbf{y}c \uplus \mathbf{y}d$, or the other way around, yielding $\mathbf{y}(c \uplus d)$, and we have $\mathbf{y}c \uplus \mathbf{y}d \to \mathbf{y}(c \uplus d)$. We can view objects of $\mathrm{Psh}(\mathcal{C})$ as objects of $\mathcal{C}$ equipped with a further, weakest, relation, and $\mathbf{y}$ can be viewed as a codiscrete embedding. Therefore, we propose to alternatively model depth p types of RelDTT in $\mathrm{Psh}^{p+2}(\mathcal{C})$ for any category $\mathcal{C}$.

Lax-Idempotent 2-Monads It is well-known that $\mathrm{Psh} : \mathsf{Cat} \to \mathsf{Cat}$ sends a category to its free cocompletion; it is then unsurprising that Psh has the structure of a (weak) 2-monad. In fact, Psh is a prototypical instance of the more general concept of a (weak) *lax-idempotent 2-monad*:

Definition 1. A (strict) 2-monad $(\mathbf{M}, \eta, \mu)$ on a (strict) 2-category $\mathbf{C}$ is **lax-idempotent** if it satisfies one of many equivalent properties [nLa23b, nLa23c, Koc95] including:

- The equality $\mathrm{id} = \mu \circ \mathbf{M}\eta$ is the unit of an adjunction $\mathbf{M}\eta \dashv \mu$,
- The equality $\mu \circ \eta\mathbf{M} = \mathrm{id}$ is the co-unit of an adjunction $\mu \dashv \eta\mathbf{M}$.

Recall that any functor $F : \mathcal{C} \to \mathcal{D}$ gives rise to a triple of adjoint functors $F_! \dashv F^* \dashv F_*$ [nLa23a]. Here, $\mathrm{Psh}\, F := F_! : \mathrm{Psh}(\mathcal{C}) \to \mathrm{Psh}(\mathcal{D})$ is taken to be the (pseudo)functorial action of Psh. The Yoneda-embedding $\eta := \mathbf{y} : \mathrm{Id} \to \mathrm{Psh}$ is (pseudo)natural and serves as the unit of the 2-monad. It turns out that the adjoint triple obtained for $F = \mathbf{y} : \mathcal{C} \to \mathrm{Psh}(\mathcal{C})$ is exactly $\mathrm{Psh}\,\eta \dashv \mu \dashv \eta\,\mathrm{Psh}$, so that Psh is indeed a (weak) lax-idempotent 2-monad.

Iterated applications of $\mathbf{M}$ generate long chains of adjoint morphisms, e.g.

$$\mathbf{MMM}\eta \dashv \mathbf{MM}\mu \dashv \mathbf{MM}\eta\mathbf{M} \dashv \mathbf{M}\mu\mathbf{M} \dashv \mathbf{M}\eta\mathbf{MM} \dashv \mu\mathbf{MM} \dashv \eta\mathbf{MMM} : \mathbf{M}^3\mathcal{C} \to \mathbf{M}^4\mathcal{C}.$$

We find similar chains in DoR,[4] listing the modalities that insert or remove 1 relation into/from the stack, which clearly generate all the morphisms of DoR:

$$\langle =, 0, 1\rangle \dashv \langle 1, 2\rangle \dashv \langle 0, 0, 1\rangle \dashv \langle 0, 2\rangle \dashv \langle 0, 1, 1\rangle \dashv \langle 0, 1\rangle \dashv \langle 0, 1, \top\rangle : 1 \to 2.$$

Proposition 2. The following defines is a lax-idempotent monad on the mode theory DoR:

$$\begin{aligned}
&\mathbf{M}\,p = p+1, && \eta : p \to p+1 \\
&i \cdot \mathbf{M}(\mu : p \to q) = \begin{cases} \quad (=) & \text{if } p = -2, \\ \text{else} \quad p+1 & \text{if } i = q+1, \\ \text{else} \quad p+1 & \text{if } i \cdot \mu = \top, \\ \text{else} \quad i \cdot \mu, & \end{cases} && \begin{aligned} &\eta = \langle 0, \ldots, p, \top\rangle \\ \\ &\mu : p+1 \to p \\ &\mu = \langle 0, \ldots, p\rangle \end{aligned}
\end{aligned}$$

Theorem 3 (WIP)**.** The 2-category DoR is *freely* generated by the object -2 and the existence of a strict lax-idempotent 2-monad $(\mathbf{M}, \eta, \mu)$.

Sketch of proof. By analysis of string diagram representations of 1-morphisms [nLa23d, nLa23e]. The relations $0, \ldots, p$ correspond to regions enclosed between two strings, while $=$ and $\top$ are the outer regions. A coherence property can be proven for commuting 2-cells around μ, thus establishing uniqueness of 2-cells. □

This makes RelDTT the internal language of a strict lax-idempotent 2-monad $\mathbf{M} : \mathsf{Cat} \to \mathsf{Cat}$ iteratively applied to a single category, assuming that $\mathbf{M}$ produces CwFs [Dyb96] and that all generated right adjoints are CwF morphisms (which is the case for Psh [Nuy20, thm. 6.4.1]). Modulo strictification, we can instantiate $\mathbf{M}$ with Psh, as desired.

[4] In fact, picking $\mathcal{C} = -2$ and $\mathbf{M}$ as in proposition 2, the chains for $\mathbf{M}$ specialize to the ones on DoR.

Acknowledgements Andreas Nuyts holds a Postdoctoral Fellowship from the Research Foundation - Flanders (FWO). This research is partially funded by the Research Fund KU Leuven.

References

[ACK16] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending homotopy type theory with strict equality. *CoRR*, abs/1604.03799, 2016. URL: http://arxiv.org/abs/1604.03799.

[ACKS17] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications, 2017. URL: https://arxiv.org/abs/1705.03307, doi:10.48550/ARXIV.1705.03307.

[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.

[Atk12] 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.

[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.

[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.

[Dyb96] Peter Dybjer. *Internal type theory*, pages 120–134. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. doi:10.1007/3-540-61780-9_66.

[GKNB21] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal dependent type theory. *Log. Methods Comput. Sci.*, 17(3), 2021. doi:10.46298/lmcs-17(3:11)2021.

[Koc95] Anders Kock. Monads for which structures are adjoint to units. *Journal of Pure and Applied Algebra*, 104(1):41–59, 1995.

[Lei91] Daniel Leivant. Finitely stratified polymorphism. *Information and Computation*, 93(1):93 – 113, 1991. doi:http://dx.doi.org/10.1016/0890-5401(91)90053-5.

[ND18] 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.

[nLa23a] nLab authors. functoriality of categories of presheaves. https://ncatlab.org/nlab/show/functoriality+of+categories+of+presheaves, March 2023. Revision 4.

[nLa23b] nLab authors. lax-idempotent 2-adjunction. https://ncatlab.org/nlab/show/lax-idempotent+2-adjunction, March 2023. Revision 7.

[nLa23c] nLab authors. lax-idempotent 2-monad. https://ncatlab.org/nlab/show/lax-idempotent+2-monad, March 2023. Revision 27.

[nLa23d] nLab authors. monad. https://ncatlab.org/nlab/show/monad, March 2023. Revision 102.

[nLa23e] nLab authors. string diagram. https://ncatlab.org/nlab/show/string+diagram, March 2023. Revision 79.

[Nuy20] Andreas Nuyts. *Contributions to Multimode and Presheaf Type Theory*. PhD thesis, KU Leuven, Belgium, 8 2020. URL: https://anuyts.github.io/files/phd.pdf.

[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.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[Rey84] John C. Reynolds. Polymorphism is not set-theoretic. Research Report RR-0296, INRIA, 1984. URL: https://hal.inria.fr/inria-00076261.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, IAS, 2013.

[Voe13] Vladimir Voevodsky. A simple type system with two identity types. unpublished note, 2013. URL: https://ncatlab.org/homotopytypetheory/files/HTS.pdf.

Conservativity of Type Theory over Higher-order Arithmetic

Benno van den Berg and Daniël Otten

University of Amsterdam, Amsterdam, The Netherlands
`bennovdberg@gmail.com` and `daniel@otten.co`

In this work we investigate how much type theories are able to prove about the natural numbers. We show that strong versions of type theory (both predicative and impredicative) prove exactly the same arithmetical formulas as Higher-order Heyting Arithmetic (HAH). As a consequence, we see that these versions are equiconsistent with HAH. Along the way, we investigate the different interpretations of higher-order logic in type theory, and to what extent dependent type theories can be seen as extensions of higher-order logic.

Main Theorem. We have the following result:

λC+ and ML1+ are conservative over HAH.

Here λC+ is a version of the Calculus of Inductive Constructions [CH88, BC13, PM15], while ML1+ is Martin-Löf type theory [ML84] with a single universe, extended with propositional truncation, resizing, and quotient types. We will explain these theories and sketch the proofs.

HAH [TvD88, Bus98] uses the same axioms for the natural numbers as Peano Arithmetic (PA), however it is formulated in intuitionistic higher-order logic instead of classical first-order logic. In higher-order logic we can quantify over powersets of the domain: we write x^n if x is an element of the n-th powerset, in our case $\mathcal{P}^n(\mathbb{N})$. This is governed by the axiom schemes:

$$\forall X^{n+1}\,\forall Y^{n+1}\,(\forall z^n\,(z \in X \leftrightarrow z \in Y) \to X = Y), \qquad \text{(extensionality)}$$

$$\exists X^{n+1}\,\forall z^n\,(z \in X \leftrightarrow P[z]). \qquad \text{(comprehension)}$$

Powersets are ubiquitous in mathematical practice, however their interpretation in type theory is not always straightforward.

Impredicative type theory has a canonical interpretation of higher-order logic, including powersets. In an impredicative version of type theory there exists a special universe Prop that is closed under products over all types. So, if we have an arbitrary type A, and for $x : A$ a type $B[x] : \mathsf{Prop}$, then we always have $\Pi(x : A)\,B[x] : \mathsf{Prop}$. We think of the types in Prop as propositions, and write $\forall(x : A)\,B[x]$ for $\Pi(x : A)\,B[x]$ in the case that $B[x] : \mathsf{Prop}$. The other logical connectives can also be defined:

$$\bot :\equiv \forall(C : \mathsf{Prop})\,C, \qquad A \vee B :\equiv \forall(C : \mathsf{Prop})\,((A \to C) \to ((B \to C) \to C)),$$
$$\top :\equiv \forall(C : \mathsf{Prop})\,(C \to C), \qquad A \wedge B :\equiv \forall(C : \mathsf{Prop})\,((A \to (B \to C)) \to C),$$

$$\mathcal{P}\,A :\equiv A \to \mathsf{Prop}, \qquad \exists(x : A)\,B[x] :\equiv \forall(C : \mathsf{Prop})\,((\forall(x : A)\,(B[x] \to C)) \to C).$$

The impredicativity allows us to quantify over propositions while defining a proposition.

Besides logic, an impredicative universe also allows us to define weak versions of inductive and coinductive data types like the natural numbers and streams, however we cannot prove that they satisfy the full (co)induction principles [Geu01]. The reason for this limitation is that an impredicative universe is able to fit two roles: it is consistent that the types in this universe have at most one distinct term [Smi88], but also that it has strong (co)inductive types [Hyl88].

λC+ makes use of this fact by postulating two impredicative universes $\mathsf{Prop}, \mathsf{Set} : \mathsf{Type}$ with different properties. To satisfy extensionality we add the following axioms for Prop:

$$\begin{aligned} \mathsf{fun\text{-}ext} &: \forall(f, f' : \Pi(x : A)\, B\,[x])\,((\forall(x : A)\,(f\,x = f'\,x)) \to (f = f')), \\ \mathsf{prop\text{-}ext} &: \forall(P, P' : \mathsf{Prop})\,((P \to P') \to (P' \to P) \to (P = P')). \end{aligned}$$

In contrast, we assume that Set has a wide array of (co)inductive types: $\mathbb{0}, \mathbb{1}, \mathbb{2}, \mathbb{N}, \Sigma, \Pi, \mathrm{W}, \mathrm{M}$, and quotient types. This gives a strong type theory with both versions of impredicativity.

Proof Sketch. (λC+ is conservative over HAH) It is straightforward to see that λC+ proves the axioms and satisfies the rules of HAH. To show that it does not prove more HAH-formulas, we give an interpretation of λC+ in HAH and show that the composition of interpretations $\mathrm{HAH} \hookrightarrow \lambda\mathrm{C}+ \to \mathrm{HAH}$ is the identity up to logical equivalence.

Our interpretation of λC+ in HAH can be seen as a model for λC+ within HAH. We interpret the propositions, sets, and types of λC+ as subsingletons, partial equivalence relations (PERs), and assemblies, respectively. This is based on existing techniques [Hyl88, Reu99], but modified in two fundamental ways: (a) we restrict the model to get an interpretation in HAH instead of Zermelo-Fraenkel set theory, and (b) we extend the model from the minimalistic Calculus of Constructions to our extensive Calculus of Inductive Constructions λC+.

Predicative type theory gives a more complicated story. This is because predicative theories, without impredicative universes like Prop, do not allow the same canonical interpretation of higher-order logic. Instead, multiple different interpretations exist in the literature, which actually affect the formulas that are provable in our type theory. First off, there are two ways to interpret logical connectives (with or without propositional truncation) [Uni13]:

$$\begin{aligned} (A \vee B)^\circ &:= \|A^\circ + B^\circ\|, & (A \vee B)^* &:= A^* + B^*, \\ (A \wedge B)^\circ &:= \ A^\circ \times B^\circ, & (A \wedge B)^* &:= A^* \times B^*, \\ (\exists(x \in A)\, B[x])^\circ &:= \|\Sigma(x : A)\, B[x]^\circ\|, & (\exists(x \in A)\, B[x])^* &:= \Sigma(x : A)\, B[x]^*, \\ (\forall(x \in A)\, B[x])^\circ &:= \ \Pi(x : A)\, B[x]^\circ, & (\forall(x \in A)\, B[x])^* &:= \Pi(x : A)\, B[x]^*. \end{aligned}$$

The second option proves the axiom of choice, which is not provable in HAH [CR12]. However, if we only consider first-order formulas, then we can still get conservativity results [Ott22]. To get conservativity for higher-order formulas as well, we focus on the first interpretation.

The more difficult problem is powersets: if we take $\mathcal{P}\,A := A \to \mathsf{Type}_0$, then we cannot prove extensionality and comprehension. Extensionality is not true because, given $X, Y : A \to \mathsf{Type}_0$ such that for $z : A$ we have functions $X\,z \to Y\,z$ and $Y\,z \to X\,z$, we do not necessarily have $X = Y$. Consider for example $X := \lambda z\, \mathbb{1}$ and $Y := \lambda z\, \mathbb{2}$. To recover extensionality, we can work with quotient types: take $\mathcal{P}_q\,A := (A \to \mathsf{Type}_0)/{\approx}$, where $(X \approx Y) := \Pi(z : A)\,(X\,z \leftrightarrow Y\,z)$. Alternatively, we could also deal with typeoids: types with an associated equivalence relation. For comprehension, the problem is that formulas cannot quantify over themselves: if a type contains a universe Type_i then it ends up in a higher universe Type_{i+1}. To counter this, we add the axiom of propositional resizing, allowing us to find equivalent types in lower universes.

ML1+ has a predicative universe $\mathsf{Type}_0 : \mathsf{Type}_1$, with $\mathbb{0}, \mathbb{1}, \mathbb{2}, \mathbb{N}, \Sigma, \Pi, \mathrm{W}, \mathrm{M}, \|\cdot\|$, and quotient types. We have an axiom for propositional resizing and interpret higher-order logic using propositional truncation and quotient types.

Proof Sketch. (ML1+ is conservative over HAH) We first embed ML1+ in λC+ by sending Type_0 to Set and Type_1 to Type. Then we show that the interpretation of logic in ML1+ is equivalent to the impredicative interpretation of logic in λC+.

References

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development, Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[Bus98] Samuel Buss. *Handbook of Proof Theory*. Elsevier, 1998.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988.

[CR12] Ray-Ming Chen and Michael Rathjen. Lifschitz realizability for intuitionistic Zermelo–Fraenkel set theory. *Archive for Mathematical Logic*, 51(7):789–818, 2012.

[Geu01] Herman Geuvers. Induction is not derivable in second order dependent type theory. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, pages 166–181, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[Hyl88] Martin Hyland. A small complete category. *Annals of pure and applied logic*, 40(2):135–165, 1988.

[ML84] Per Martin-Löf. *Intuitionistic type theory*. Studies in proof theory. Lecture notes; 1 861180607. Bibliopolis, Napoli, 1984.

[Ott22] Daniël Otten. De Jongh's theorem for type theory. Master's thesis, University of Amsterdam, 2022.

[PM15] Christine Paulin-Mohring. Introduction to the calculus of inductive constructions, 2015.

[Reu99] Bernhard Reus. Realizability models for type theories. *Electronic Notes in Theoretical Computer Science*, 23(1):128–158, 1999. Tutorial Workshop on Realizability Semantics and Applications (associated to FLoC'99, the 1999 Federated Logic Conference).

[Smi88] Jan M. Smith. The independence of Peano's fourth axiom from Martin-Löf's type theory without universes. *The Journal of Symbolic Logic*, 53(3):840–845, 1988.

[TvD88] Anne Sjerp Troelstra and Dirk van Dalen. *Constructivism in Mathematics*, volume I. North-Holland Publishing Co., 1988.

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

Enriched Categories in Univalent Foundations

Niels van der Weide

Radboud Universiteit, Nijmegen, The Netherlands
nweide@cs.ru.nl

Enriched categories have found numerous applications, including effects in programming languages [EMS14, PP01], abstract homotopy theory [GJ09], and in higher category theory [Lur09]. In this abstract, we discuss an ongoing formalization of enriched categories in univalent foundations. More specifically, we define the notion of univalence for enriched categories, and we prove that the bicategory of univalent enriched category is univalent. This gives us a structure identity principle for enriched categories. The definitions and theorems in this abstract are formalized in the Coq proof assistant [Tea22] using the UniMath library [VAG+] and building upon [AKS15, WMA22], and all our definitions and theorems are available in UniMath[1].

1 Univalent Enriched Categories

In the remainder of this abstract, we fix a monoidal category $\mathcal{V}$, and we denote its unit by $\mathbb{1}$ and the tensor by $\otimes$. Usually, the definition of an enriched category is a slight modification of the notion of category: the homs are required to be objects of $\mathcal{V}$ instead of sets. However, we take a different approach, which is based on the notion of *enrichment*. Since every enriched category C has an underlying category C_0, there is a 2-functor $(-)_0$ from the 2-category $\mathcal{V}\mathsf{Cat}$ of categories enriched over $\mathcal{V}$ to the 2-category Cat of categories [Kel82]. The idea is that an enrichment of C is an object of the fiber of $(-)_0$ along C.

Definition 1. A **$\mathcal{V}$-enrichment** E of a category C consists of

- a function $\mathsf{E}(-,-) : \mathsf{C} \to \mathsf{C} \to \mathcal{V}$;
- for all $x : \mathsf{C}$ a morphism $\mathsf{Id} : \mathbb{1} \to \mathsf{E}(x,x)$ in $\mathcal{V}$;
- for all $x, y, z : \mathsf{C}$ a morphism $\mathsf{Comp} : \mathsf{E}(y,z) \otimes \mathsf{E}(x,y) \to \mathsf{E}(y,z)$ in $\mathcal{V}$;
- functions $\mathsf{FromArr} : \mathsf{C}(x,y) \to \mathcal{V}(\mathbb{1}, \mathsf{E}(x,y))$ and $\mathsf{ToArr} : \mathcal{V}(\mathbb{1}, \mathsf{E}(x,y)) \to \mathsf{C}(x,y)$ for all $x, y : \mathsf{C}$

such that the usual axioms for enriched categories are satisfied and such that $\mathsf{FromArr}$ and ToArr are inverses of each other. A **category with a $\mathcal{V}$-enrichment** is a pair of a category C together with a $\mathcal{V}$-enrichment of C.

Note that enrichments of categories have been considered in other work as well [MU22], although they used yet another definition. The reason why we choose to define enriched categories this way, is because using enrichments, we can define a displayed bicategory $\mathcal{V}\mathsf{UnivCat}_{\mathsf{disp}}$ over $\mathsf{UnivCat}$ whose total bicategory is the bicategory $\mathcal{V}\mathsf{UnivCat}$ of enriched categories (Definition 4). This way the proof of the univalence for the bicategory of enriched categories becomes simpler, because we can reuse the proof that the bicategory of categories is univalent [AFM+21]. Note that our notion of categories with a $\mathcal{V}$-enrichment is actually equivalent to the usual notion of enriched categories.

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

Proposition 2. *The type of categories with a $\mathcal{V}$-enrichment is equivalent to the type of $\mathcal{V}$-enriched categories defined using the definition given by Kelly [Kel82].*

Next we define *univalent enriched categories.* With our definition of enrichments, we say that a univalent enriched category is a univalent category together with an enrichment. Equivalently, we also phrase univalence for enriched categories as defined in [Kel82]: such an enriched category C would be univalent if the underlying category C_0 is univalent.

Definition 3. A **univalent $\mathcal{V}$-enriched category** is a pair of a univalent category C together with a $\mathcal{V}$-enrichment of C.

2 The Bicategory of Univalent Enriched Categories

Next we construct the bicategory of univalent enriched categories, and we prove that this bicategory is univalent. To define this bicategory, we use displayed bicategories [AFM$^+$21], and thus we need to define enrichments for functors and natural transformations. Concretely, we need to define $\mathcal{V}$-enrichments for functors $\mathsf{F} : \mathsf{C}_1 \to \mathsf{C}_2$ from E_1 to E_2, where E_1 and E_2 are $\mathcal{V}$-enrichments of C_1 and C_2 respectively. We also need to define $\mathcal{V}$-naturality for natural transformations. The definitions of these notions are in a similar style as Definition 1, and for the precise definitions, we refer the reader to the formalization.

Definition 4. We define the displayed bicategory $\mathcal{V}\mathsf{UnivCat}_{\mathsf{disp}}$ over $\mathsf{UnivCat}$ as follows:

- The displayed objects over a category C are $\mathcal{V}$-enrichments of C;
- The displayed 1-cells over a functor $\mathsf{F} : \mathsf{C}_1 \to \mathsf{C}_2$ from E_1 to E_2 are $\mathcal{V}$-enrichments of F;
- The displayed 2-cells over a natural transformation τ are proofs that τ is $\mathcal{V}$-natural.

The total bicategory of $\mathcal{V}\mathsf{UnivCat}_{\mathsf{disp}}$ is the bicategory of univalent enriched categories, and we denote it by $\mathcal{V}\mathsf{UnivCat}$.

The proof that the data in Definition 4 actually forms a displayed bicategory, is similar to the construction of the bicategory of enriched categories in set-theoretic foundations. We conclude this abstract by proving that $\mathcal{V}\mathsf{UnivCat}$ is univalent.

Lemma 5. *If $\mathcal{V}$ is univalent, then the displayed bicategory $\mathcal{V}\mathsf{UnivCat}_{\mathsf{disp}}$ is univalent.*

Theorem 6. *If $\mathcal{V}$ is univalent, then the bicategory $\mathcal{V}\mathsf{UnivCat}$ is univalent.*

The methods used to prove Theorem 6 are similar to the methods used for proofs of univalence in [AFM$^+$21]. Concretely, this theorem says that two enriched categories are equal if we have an enriched equivalence between them. As such, we obtain a structure identity principle for enriched categories.

There are numerous way to extend the work in this abstract. One particular way, is by instantiating the formal theory of monads to enriched categories [Str72, vdW22]. Concretely, this means that one constructs the enriched Eilenberg-Moore and Kleisli category for an enriched monad. The usual theorems about monads and adjunctions for enriched categories would then follow from the formal theory developed by Street [Str72], and these theorems are useful for formalizing results about the semantics of the extended effect calculus [EMS14].

References

[AFM$^+$21] Benedikt Ahrens, Dan Frumin, Marco Maggesi, Niccolò Veltri, and Niels van der Weide. Bicategories in univalent foundations. *Math. Struct. Comput. Sci.*, 31(10):1232–1269, 2021.

[AKS15] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Mathematical Structures in Computer Science*, 25(5):1010–1039, 2015.

[EMS14] Jeff Egger, Rasmus Ejlers Møgelberg, and Alex Simpson. The enriched effect calculus: syntax and semantics. *J. Log. Comput.*, 24(3):615–654, 2014.

[GJ09] Paul G Goerss and John F Jardine. *Simplicial homotopy theory*. Springer Science & Business Media, 2009.

[Kel82] Max Kelly. *Basic concepts of enriched category theory*, volume 64. CUP Archive, 1982.

[Lur09] Jacob Lurie. *Higher topos theory*. Princeton University Press, 2009.

[MU22] Dylan McDermott and Tarmo Uustalu. What makes a strong monad? In Jeremy Gibbons and Max S. New, editors, *Proceedings Ninth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2022, Munich, Germany, 2nd April 2022*, volume 360 of *EPTCS*, pages 113–133, 2022.

[PP01] Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.

[Str72] Ross Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2(2):149–168, 1972.

[Tea22] The Coq Development Team. The Coq Proof Assistant, January 2022.

[VAG$^+$] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. Unimath — a computer-checked library of univalent mathematics. available at `http://unimath.org`.

[vdW22] Niels van der Weide. The Formal Theory of Monads, Univalently. *CoRR*, abs/2212.08515, 2022.

[WMA22] Kobe Wullaert, Ralph Matthes, and Benedikt Ahrens. Univalent monoidal categories. *arXiv preprint arXiv:2212.03146*, 2022.

4

Session 5: Links between type theory and functional programming

Normal Form Bisimulations by Value

Beniamino Accattoli, Adrienne Lancelot, and Claudia Faggian

The study of program equivalences for λ-calculi is an important topic where semantical and operational techniques meet. Properties of program equivalences are notoriously difficult to prove. Even the equivalence of two terms might be challenging to establish, if the notion of equivalence is Morris' contextual equivalence [Mor68], or some variant still quantifying over a class of contexts, such as Abramsky's applicative bisimilarity [Abr90], as opposed to bisimulations on finitely branching transition systems [DGHL09]. Another difficulty is the fact that properties of program equivalences are quite brittle, as they are not preserved by extensions of the calculus under study, nor by restrictions, and not even by changing the evaluation strategy in the calculus.

This paper stems from the observation that another natural program equivalence, Sangiorgi's *normal form bisimilarity* [San94] (shortened to nf-bisimilarity), behaves differently in call-by-name (shortened to CbN) and call-by-value (CbV), already in the untyped effect-free weak case. We then study various CbV nf-bisimilarities and, to better understand them, we relate them to Ehrhard's CbV relational model [Ehr12] that we present via non-idempotent intersection types.

Normal Form Bisimilarity. Normal form bisimulations are program equivalences that, instead of comparing terms *externally*, depending on how they behave *in contexts*, compare them *internally*, by looking at the structure of their (infinitary) *normal forms*. A distinctive feature of nf-bisimulations is that they directly manipulate *open terms*, to the point that Sangiorgi rather used to call them *open bisimulations* in his seminal paper [San94].

It is known that Sangiorgi's CbN nf-bisimilarity is not fully abstract for contextual equivalence, being sound but not complete. The failure of full abstraction is compensated by the fact that CbN nf-bisimilarity is easier to establish than applicative bisimilarity, because of the absence of quantification over arguments. Typically, it is easy to show that different fix-points combinators—which are the paradigmatic terms with infinitary normal forms—are nf-bisimilar, while it is hard to show that they are applicative bisimilar.

In CbV, however, it is not so obvious that contextual equivalence $\simeq_C^v$ should be the standard of reference, at least in the untyped, effect-free setting, because therein $\simeq_C^v$ is *cost-insensitive*: it equates terms such as $(\lambda x.yxx)t$ and ytt, for any t, also for terms t that are not values. This is against the very idea of CbV, of avoiding duplicating t before having evaluated it. In richer CbV settings with state or probability, contexts discriminate more, and those terms are separated, but in the pure case they are not.

Meaninglessness. Idle programs which diverge in an unproductive way are sometimes called *meaningless*. A key fact is that all meaningless terms are contextually equivalent to $\Omega = \delta\delta$ where $\delta = (\lambda x.xx)$, the paradigmatic meaningless term.

In CbV, meaningless terms should not be defined by simply changing the evaluation strategy in their CbN definition (that is, Barendregt's unsolvability [Bar84]), as this approach yields a class of terms which are not all contextually equivalent in CbV [AG22]. The weak variant of CbV unsolvable terms, called *inscrutable terms* (or *non-potentially-valuable* by Paolini and Ronchi della Rocca [PRDR99, RP04]) provides the right semantic foundation in CbV.

It turns out, unfortunately, that inscrutable terms still lack some expected properties in Plotkin's CbV calculus, namely they do *not* all diverge. Such an issue entails that *any* nf-bisimulation based on Plotkin's calculus fails at being complete, because it does not equate meaningless terms. A possible way out is to switch to an extension of Plotkin's CbV calculus where CbV inscrutable terms do all diverge while preserving the same notion of contextual equivalence, and design therein a nf-bisimilarity. One such setting is the *value substitution calculus* (shortened to VSC), a CbV λ-calculus due to Accattoli and Paolini [AP12], related to linear logic proof nets [Acc15] and where inscrutability has been extensively studied [AP12, AG22].

A CbV Nf-Bisimilarity $\simeq_{net}$ Equating Meaningless Terms The motivation behind this work is the development of a CbV nf-bisimilarity that equates CbV inscrutable terms, aiming at refining known CbV program equivalences and matching Sangiorgi's CbN nf-bisimilarity at the same time. By using the VSC, we do build a CbV nf-bisimilarity matching Sangiorgi's in capturing inscrutable terms. The obtained *net bisimilarity* $\simeq_{net}$ and the proof of its *compatibility* (that is, stability by context closure)—which is the challenging property to prove for bisimilarities—are the main contributions of this paper. Compatibility implies soundness with respect to contextual equivalence, and it is proved adapting Lassen's variant for nf-bisimilarities of Howe's method [Las99]. As it is often the case for nf-bisimilarities, ours is sound but not complete, and it is *cost-sensitive.*

The crafting of net bisimilarity is based on a sophisticated analysis of CbV and the VSC. In particular, its definition compares normal forms modulo some equivalences induced by linear logic proof nets, whence the name *net* bisimilarity. We actually go further, introducing a *parametric* nf-bisimilarity, where such extra equivalences can be turned off and on at will—because some fail in extensions of CbV with effects—thus defining a *family* of CbV nf-bisimilarity, all proved compatible via a single abstract proof.

Our result is however more a new beginning than the end of the story: net bisimilarity, indeed, is *not* a refinement of the state of the art for CbV nf-bisimilarity, namely Lassen's enf bisimilarity $\simeq_{enf}$ [Las05]. In fact, the two are *incomparable.* An important point is that Lassen's program equivalence follows Moggi's extension of the CbV calculi [Mog88, Mog89]: in particular, it verifies the *left identity law* $\mathtt{I}t \equiv_{lid} t$, where $\mathtt{I} = \lambda x.x$. If t is a value, the law is included in β_v-reduction, but Moggi extends it to *every term t*. Moggi's left identity law, however, is not a rule of the VSC and net bisimilarity does not verify it.

Type Equivalence $\simeq_{type}$ = Meaningless and Left Identity Normal form bisimulations are operationally-based equivalences. Denotational semantics also yield equational theories—by equating terms with the same interpretation—which are program equivalences. Such model-based equivalences differ from nf-bisimulations. On the one hand, they are easily proved compatible, while nf-bisimulations are not. On the other hand, as contextual equivalence, they are not directly *usable*, because the interpretation of a term usually contains infinitely many elements.

Here we investigate the equational theory of Ehrhard's CbV relational model [Ehr12]. We call it *type equivalence* because the model is presented as a multi type system (a variant of intersection types). Such a model was already extensively studied in connection with the VSC by Accattoli and Guerrieri [AG18, AGL21, AG22]. Its equational theory does not have a presentation via nf-bisimulations, nor any other characterization, but it is nonetheless possible to study it via the multi type system. It turns out that type equivalence, similarly to nf-bisimilarities, is compatible and sound, but not complete for contextual equivalence, as it is cost-sensitive.

We prove that Lassen's enf bisimilarity and net bisimilarity are *included* in type equivalence. Therefore, the two bisimilarities are joinable. Since both are sound, they are obviously joinable in a cost-insensitive setting, as they are both included in contextual equivalence. Our results show that they are also joinable in a *cost-sensitive* program equivalence, thus suggesting that a nf-bisimilarity joining the two might be possible. Crafting it, and especially proving that it is compatible and that it includes type equivalence, is left to future work.

References

[Abr90] Samson Abramsky. *The Lazy Lambda Calculus*, page 65–116. Addison-Wesley Longman Publishing Co., Inc., USA, 1990.

[Acc15] Beniamino Accattoli. Proof nets and the call-by-value λ-calculus. *Theor. Comput. Sci.*, 606:2–24, 2015.

[AG18] Beniamino Accattoli and Giulio Guerrieri. Types of fireballs. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 45–66. Springer, 2018.

[AG22] Beniamino Accattoli and Giulio Guerrieri. The theory of call-by-value solvability. *Proc. ACM Program. Lang.*, 6(ICFP):855–885, 2022.

[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 Tom Schrijvers and Peter Thiemann, editors, *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*, volume 7294 of *Lecture Notes in Computer Science*, pages 4–16. Springer, 2012.

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1984.

[DGHL09] Pietro Di Gianantonio, Furio Honsell, and Marina Lenisa. Rpo, second-order contexts, and lambda-calculus. *Logical Methods in Computer Science*, 5, 06 2009.

[Ehr12] Thomas Ehrhard. Collapsing non-idempotent intersection types. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 259–273. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[Las99] Søren B. Lassen. Bisimulation in untyped lambda calculus: Böhm trees and bisimulation up to context. 20:346–374, 1999.

[Las05] Soren Lassen. Eager normal form bisimulation. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, LICS '05, page 345–354, USA, 2005. IEEE Computer Society.

[Mog88] Eugenio Moggi. Computational λ-Calculus and Monads. LFCS report ECS-LFCS-88-66, University of Edinburgh, 1988.

[Mog89] Eugenio Moggi. Computational λ-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23. IEEE Computer Society, 1989.

[Mor68] James Hiram Morris. *Lambda-calculus Models of Programming Languages.* PhD thesis, Massachusetts Institute of Technology, 1968.

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

[San94] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, 1994.

Consequences of the modal unification of the functional calling paradigms

José Espírito Santo[1], Dylan McDermott[2], 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 modal unification of the calling paradigms call-by-name (cbn) and call-by-value (cbv) carried out in [2, 3] follows these general lines: (1) The embeddings of intuitionistic logic into modal logic S4 attributed to Girard and Gödel [7] are recast as maps compiling respectively the ordinary, cbn λ-calculus [1] and Plotkin's cbv λ-calculus [6] into a modal target, which is a simple extension of the λ-calculus with an S4 modality, denoted $\Box$. (2) One can define later instantiations of the S4 modality, in the form of interpretations of the modal target into diverse other calculi, like the linear λ-calculus [5] or call-by-push-value [4], recovering by composition known embeddings of cbn and cbv λ-calculus, and confirming the thesis that calculi encompassing call-by-name and call-by-value do so because they already embed our modal target.

In this abstract we report on-going work on the instantiation into call-by-push-value, which was done using the modal target $\lambda_{\glj}$ introduced in [3].

In the $\lambda_{\glj}$-calculus, the type form $\Box A$ is permitted, if A itself is not a modal type. This is implemented by splitting types A into boxed types B and unboxed types C:

$$A ::= B \,|\, C \qquad B ::= \Box C \qquad C ::= X \,|\, B \supset A$$

In contexts Γ, variables will be assigned boxed types only. Accordingly, implications have the form $B \supset A$. The splitting of the types almost induces a splitting of the terms, that is, a classification of each term form as either boxed or unboxed, in the sense of being necessarily typed with a boxed or unboxed type. For instance, the constructor $\mathsf{box}(M)$ corresponding to the introduction of $\Box$ is boxed, while $\varepsilon(x)$ is unboxed (variables and the eliminator of the modality will always occur together in this compound form). Only the application constructor as to be split into a boxed and a unboxed variants. Summing up, terms T are either boxed terms P or unboxed terms M, given by:

$$M, N ::= \varepsilon(x) \,|\, \lambda x.T \,|\, MQ \qquad\qquad P, Q ::= \mathsf{box}(M) \,|\, QP$$

The second application form QP has type B, if $P : \Box(B' \supset B)$ and $Q : B'$; so P is the function and Q is the argument. Hence, the head of a unboxed term $M = VQ_1 \cdots Q_m$ is a value (here $\varepsilon(x)$ or $\lambda x.T$), while the head of a boxed term $P = Q_m \cdots Q_1\mathsf{box}(M)$ is a box. The remaining typing rules are easily guessed (and given in [3]). To conclude the definition of $\lambda_{\glj}$, we state its two reduction rules:

$$(\beta_<) \quad (\lambda x.M)\mathsf{box}(N) \to [N/\varepsilon(x)]M \qquad (\beta_>) \quad \mathsf{box}(N)\mathsf{box}(\lambda x.P) \to [N/\varepsilon(x)]P$$

System $\lambda_{\glj}$ was designed from intuitionistic $S4$, through several stages of simplification [2, 3], aiming for a system containing as tightly as possible the images of both modal embeddings. A measure of tightness is that both cbn and cbv λ-calculus can be recognized inside $\lambda_{\glj}$: the first as the subsystem of unboxed terms, where QP and $\beta_>$ are forbidden, and boxed terms are only used as the arguments Q in application MQ, and so can be dispensed with; the second as the subsystem of boxed terms, where MQ and $\beta_<$ are forbidden, and unboxed terms are only used as values in $\mathsf{box}(V)$. Girard's (resp. Gödel's) embedding is thus the isomorphism between cbn (resp. cbv) λ-calculus and its copy inside $\lambda_{\asymp}$.

The translation of $\lambda_{\asymp}$ into call-by-push-value uses the fragment λ_{cbpv} of the latter, containing implication, F and U as the only type operations. The types of λ_{cbpv} are thus computation types A, which can be $B \supset A$, FB, or an atom X, or value types B, with unique form UA. The terms M and values V of λ_{cbpv} are given by:

$$M, N ::= \lambda x.M \mid MV \mid \mathsf{return}\, V \mid \mathsf{force}\, V \mid M \,\mathsf{to}\, x.N \qquad\qquad V, W ::= x \mid \mathsf{thunk}\, M$$

Regarding typing, terms (resp. values) receive computation (resp. value) types, under contexts Γ assigning value types to variables. Constructors $\mathsf{thunk}\, M$ and $\mathsf{force}\, V$ correspond to the introduction and elimination of U. Constructors $\mathsf{return}\, V$ and $M \,\mathsf{to}\, x.N$ correspond to the introduction and elimination of F. Each type former $\supset$, F and U has its own β-rule.

Roughly speaking, the translation $(\_)^{\triangledown} : \lambda_{\asymp} \to \lambda_{\mathsf{cbpv}}$ implements the instantiation $\Box = FU$. More precisely, there is a translation of types so that $A^{\triangledown}$ is a computation type given by $X^{\triangledown} = X$, $(B \supset A)^{\triangledown} = B^{\rhd} \supset A^{\triangledown}$, and $B^{\triangledown} = FB^{\rhd}$. Here $B^{\rhd} = (\Box C)^{\rhd}$ is the value type $UC^{\triangledown}$. Hence, $(\Box C)^{\triangledown} = FUC^{\triangledown}$. There is a translation of terms so that a unboxed term $M : C$ (resp. a boxed term $P : B$) under Γ is translated to a term $M^{\triangledown} : C^{\triangledown}$ (resp. $P^{\triangledown} : B^{\triangledown}$) under $\Gamma^{\rhd}$. The two forms of applications are translated thus:

$$(MQ)^{\triangledown} = Q^{\triangledown} \,\mathsf{to}\, x.M^{\triangledown}x \qquad\qquad (QP)^{\triangledown} = P^{\triangledown} \,\mathsf{to}\, f.Q^{\triangledown} \,\mathsf{to}\, x.(\mathsf{force}\, f)x$$

In fact, some optimizations are possible in this definition, if P or Q have the form $\mathsf{box}(M)$. We adopt such an optimization right away in the first equation, separating the case $(M\mathsf{box}(N))^{\triangledown} = M^{\triangledown}(\mathsf{thunk}\, N^{\triangledown})$.

Stabilizing the definition like that, the instantiation $(\_)^{\triangledown}$ obtains a simulation of reduction in $\lambda_{\asymp}$ by reduction in λ_{cbpv}. Moreover, recall $\lambda_{\asymp}$ includes the cbn and cbv λ-calculi. Hence, if we calculate the composition of the inclusions in $\lambda_{\asymp}$ – that is, the modal embeddings – with the instantiation, then we obtain translations from the the cbn and cbv λ-calculi into λ_{cbpv}. It turns out that such translations are Levy's translations given in [4] (modulo one η-expansion in the cbn translation of variables). Levy's cbn and cbv translations can thus be factored into one of the modal embeddings and a common factor which is the instantation $\Box = FU$. We may conclude that λ_{cbpv} subsumes cbn and cbv because it interprets the call-by-box calculus $\lambda_{\asymp}$.

Like in our study of the modal embeddings, we are carefully studying the image of the instantiation map. This will give, simultaneously: a decomposition of $\lambda_{\asymp}$ induced by the decomposition $\Box = FU$, and an interesting fragment of λ_{cbpv}. A simple example of this simultaneous benefit is the following grammar of types, which defines the type structure in the referred image:

$$A ::= FB \mid C \qquad B ::= UC \qquad C ::= X \mid B \supset A$$

On the one hand, it refines the type structure of $\lambda_{\asymp}$ given above, with the modality split into two different classes of types. On the other hand, the class of such types A is a subset of the computation types of λ_{cbpv}, because here we cannot form the value type UFB. The splitting of types A into FB and C almost induces a classification of the (computation) terms of λ_{cbpv} into the *returning computation terms* P, which have type FB, and the *resulting computation terms*, which have type C. Again, only the application constructor is ambiguous and has to be given in two forms, to enforce fully such classification. The resulting syntax is both a refinement of the term syntax of $\lambda_{\asymp}$ and the *call-by-thunk* fragment of λ_{cbpv}.

References

[1] H. Barendregt, The Lambda Calculus, Vol. 103 of Studies in Logic and the Foundations of Mathematics, North-Holland, 1984.

[2] 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.

[3] 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.

[4] P. B. Levy, Call-by-push-value: Decomposing call-by-value and call-by-name, High. Order Symb. Comput. 19 (4) (2006) 377–414.

[5] J. Maraist, M. Odersky, D. N. Turner, P. Wadler, Call-by-name, call-by-value, call-by-need and the linear lambda calculus, Theoretical Compututer Sci. 228 (1–2) (1999) 175–210.

[6] G. Plotkin, Call-by-name, call-by-value and the λ-calculus, Theoretical Compututer Science 1 (1975) 125–159.

[7] A. Troelstra, H. Schwichtenberg, Basic Proof Theory, 2nd Edition, Vol. 43 of Cambridge Tracts in Theoretical Computer Science, Cambridge Univ. Press, 2000.

Modal Types for Asynchronous FRP

Patrick Bahr and Rasmus Ejlers Møgelberg

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

Reactive programs are programs that engage in an ongoing dialogue with their environment, often without ever terminating. Examples include GUIs, servers and control software. These programs are often written in imperative languages using a combination of complex features such as shared state and call-backs, which makes them error-prone and hard to read and reason about.

The idea of Functional Reactive Programming (FRP) [5] is to represent reactive programs as functions on a type of signals in a functional programming language, allowing programs to be written in a modular way on a high level of abstraction. However, care must be taken when designing languages for FRP, to ensure that all programs can be implemented in an efficient way. In particular, the type system should ensure that all programs are causal (current output does not depend on future input) and free of implicit space- and time leaks (causing the program to eventually run out of space or slow down). Often one would also like programs to be productive, in the sense that each step of the program terminates. A naive encoding of signals as streams, i.e., coinductive solutions to $\mathsf{Sig}\, A \cong A \times \mathsf{Sig}\, A$ will cause causality to fail.

Modal FRP

Recently, a number of languages have been proposed using modal types to ensure these properties [7, 6, 10, 11, 9, 2, 1, 8]. The most important of these modalities is $\bigcirc$ encoding a notion of time step. Defining the signal type to satisfy $\mathsf{Sig}\, A \cong A \times \bigcirc(\mathsf{Sig}\, A)$ stating that the tail is only available in the next time step, type checking can ensure that all programs are causal. Often this is combined with a variant of Nakano's fixed point operator [12] of type $(\bigcirc A \to A) \to A$ allowing programmers to write recursive programs while maintaining productivity. Other modal operators include $\Box$ used for stable data (data that can be kept across time steps without causing space leaks), and $\Diamond$ (eventually), which can often be encoded. These suggest a Curry-Howard correspondence [6, 8, 3] with Linear Temporal Logic [13].

In these languages $\bigcirc$ is read as a delay on a global clock. For some applications, however, the idea of a global clock is unnatural, and may lead to leaky abstractions as well as inefficient implementations. Consider, for example, a GUI application reacting to three signals of mouse coordinates, mouse clicks, and keyboard input, respectively. The global clock will have to tick whenever one of these signal is updated, even if other signals are not. One way to encode this is to work with signals of type Maybe, but this is not only unnatural, but also inefficient in many of the languages mentioned above, because the application will have to check for input on each time step.

Async RaTT

In this talk we will present a new language called Async RaTT for asynchronous modal FRP. Typing judgements are relative to a context Δ of channels for asynchronous input signals. For example, Δ may state that there are input channels $\mathsf{keyPressed} : \mathsf{Nat}$, $\mathsf{mouseCoord} : \mathsf{Nat} \times \mathsf{Nat}$, and $\mathsf{mouseButton} : 1 + 1$ (for left and right mousebutton). We refer to a subset of the input channels as a *clock* and the arrival of an input on one of the channels in a clock θ as a *tick* on the clock θ. The central new component of Async RaTT is a modality $\textcircled{$\exists$}$ for asynchronous delays. A value of type $\textcircled{$\exists$}A$ is a pair whose first component is a clock θ, and whose second component is a computation which can be evaluated to a value of type A at the time of the first tick on the clock θ. Using $\textcircled{$\exists$}$, one can define a type of asynchronous signals that satisfies $\mathsf{Sig}\, A \cong A \times \textcircled{$\exists$}(\mathsf{Sig}\, A)$. Note that this means that the clock associated with the tail of a signal may change from one step of execution to another. This is important because it allows for dynamic changes to the dataflow graph through operators such as

$$\mathsf{switch} : \mathsf{Sig}\, A \to \textcircled{$\exists$}(\mathsf{Sig}\, A) \to \mathsf{Sig}\, A$$

which returns a signal that behaves as its first input until a new signal arrives on the second input.

To avoid space leaks, Async RaTT does not allow arbitrary data to be stored from one time step to the next. One consequence of this is that $\textcircled{$\exists$}$ is not a applicative functor: In the type $\textcircled{$\exists$}(A \to B) \to \textcircled{$\exists$}A \to \textcircled{$\exists$}B$, the delayed function and input may arrive at different times, and so one would need to be stored. Instead, Async RaTT offers a synchronisation primitive

$$\mathsf{sync} : \textcircled{$\exists$}A_1 \to \textcircled{$\exists$}A_2 \to \textcircled{$\exists$}((A_1 \times \textcircled{$\exists$}A_2) + (A_2 \times \textcircled{$\exists$}A_1) + (A_1 \times A_2))$$

whose result is delayed on the union of the clocks for the input. Among other things, sync can be used to encode switch.

Aside from $\textcircled{$\exists$}$, Async RaTT uses two other modal type operators: $\Box$ (for stable data) and $\textcircled{$\forall$}$. The latter of these is used to classify data that is available at any time in the future, but not now. It is primarily used in the type of the fixed point operator

$$\mathsf{fix} : \Box(\textcircled{$\forall$}A \to A) \to A$$

The input to fix needs to be a stable function (hence the use of $\Box$) because it can be called at any time in the future. The use of $\textcircled{$\forall$}$ ensures that the recursive call happens in the next time step, thus ensuring termination of each step of computation.

Operational semantics and results

A complete Async RaTT program is a term of type $\mathsf{Sig}\, B_1 \times \cdots \times \mathsf{Sig}\, B_n$ in some context of input channels Δ. The operational semantics maps a complete program to a machine that takes asynchronous input from Δ and produces output on the n channels of types $B_1, \ldots, B_n$ respectively. The machine associates, at each step of execution, a clock to each output signal, corresponding to the first component of an element of the type $\textcircled{$\exists$}(\mathsf{Sig}\, B_i)$ of the tail of the i'th output signal. Using this, the machine can, upon the arrival of an input, decide which output signals need to be updated as a consequence of the input arriving.

The machine uses a store for delayed computations and input data. At the end of each step of execution, all delayed computations that could potentially be run in the current step are deleted from the store, and all old input data is also deleted. We show safety of this aggresive garbage collection technique, which can be understood as the machine being free of implicit space leaks [9]. We also show causality and productivity.

The talk

The talk will focus on presenting the intuitions for the modal types as well as their typing rules and a few programming examples. If time permits, we will also sketch the machine and the logical relation used for proving the operational results mentioned. The results presented here are detailed in our recent manuscript [4].

References

[1] Patrick Bahr. Modal frp for all: Functional reactive programming without space leaks in haskell. *Journal of Functional Programming*, 32:e15, 2022.

[2] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. Simply ratt: a fitch-style modal calculus for reactive programming without space leaks. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–27, 2019.

[3] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. Diamonds are not forever: Liveness in reactive programming with guarded recursion. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021.

[4] Patrick Bahr and Rasmus Ejlers Møgelberg. Asynchronous modal FRP. Manuscript, 2023. URL: https://arxiv.org/abs/2303.03170.

[5] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM.

[6] Alan Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In Koen Claessen and Nikhil Swamy, editors, *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*, pages 49–60, Philadelphia, PA, USA, 2012. ACM.

[7] Alan Jeffrey. Functional reactive types. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 54:1–54:9, New York, NY, USA, 2014. ACM.

[8] Wolfgang Jeltsch. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electronic Notes in Theoretical Computer Science*, 286:229–242, 2012.

[9] Neelakantan R. Krishnaswami. Higher-order Functional Reactive Programming Without Spacetime Leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 221–232, Boston, Massachusetts, USA, 2013. ACM.

[10] Neelakantan R. Krishnaswami and Nick Benton. Ultrametric semantics of reactive programs. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 257–266, Washington, DC, USA, June 2011. IEEE Computer Society.

[11] Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. Higher-order functional reactive programming in bounded space. 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 45–58, Philadelphia, PA, USA, 2012. ACM.

[12] Hiroshi Nakano. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 255–266, Washington, DC, USA, June 2000. IEEE Computer Society.

[13] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, USA, 1977. IEEE Computer Society.

5

Session 6: Meta-theoretic studies of type systems

A Logical Framework for Computational Type Theories with Erased Syntax and Bidirectional Typing

Thiago Felicissimo

Université Paris-Saclay, INRIA project Deducteam, LMF, ENS Paris-Saclay
thiago.felicissimo@inria.fr

Logical Frameworks (LFs) are languages for defining logical theories. If in one hand they can be used for studying theories in a unified setting, they are also of practical interest. Indeed, an implementation of a LF yields a typechecker for its theories, which can be used for prototyping with new proposals, or also rechecking proofs coming from proof assistants (as done in the Dedukti [2] project). LFs can be classified according to two categories:

Pure LFs come with a fixed definitional equality, so theories are specified by only constants. Because most type theories feature a non-trivial definitional equality, it is impossible to define such theories directly, so instead one encodes its judgment derivations [9]. This approach has the advantage that, because typechecking is kept decidable and terms of the framework represent judgment derivations of the encoded theory, implementations of such frameworks can be used for formalizing metatheory, such as in TWELF [10, 7]. On the other hand, such an encoding does not yield a typechecker for terms of the type theory, but for its judgment derivations.

Equational LFs [8, 16] allow for an extensible definitional equality, enabling the direct definition of type theories instead of their judgments derivations. But because of the need of deciding arbitrary equalities, they are in general not designed to be implemented.

If the generality of the equalities is a challenge for implementations, a way out of this problem is to restrict the accepted ones. This is the approach adopted by the logical framework DEDUKTI [2, 3], which only supports *computational* theories — that is, whose definitional equality is generated only by rewrite rules. The advantage of this design choice is that rewriting makes it easy to decide the definitional equality in a theory-agnostic way. Experiences on typechecking big libraries of proofs in DEDUKTI confirm that this can be done efficiently [5, 11]. However, DEDUKTI encodings are polluted by bureaucratic terms which need to be quontiented out in the adequacy theorem [6]. For instance, the framework terms $\lambda\ (x.@\ t\ x)$, $\lambda\ (@\ t)$ and $\lambda\ ((z.z)\ (@\ t))$ (ignoring type annotations) all represent the same object term $\lambda x.tx$, but only $\lambda\ (x.@\ t\ x)$ would be in the image of the translation function. Moreover, only theories with fully annotated syntax are accepted — for instance, one has $\langle a, b\rangle_{A,x.B}$ instead of $\langle a, b\rangle$. This does not only impact performance, user experience and complicate adequacy proofs, but also means that the goal of representing the usual syntax of theories is unfortunately not fully achieved.

1st contribution In this work we present COMPLF. Like DEDUKTI, it allows to define type theories directly and typecheck terms in them, and unlike TWELF it is not aimed at mechanizing their metatheory. Like DEDUKTI, our framework only supports computational theories, which makes equality checking easy. However, in COMPLF bureaucratic terms such as $\lambda\ (@\ t)$ and $\lambda\ ((z.z)\ (@\ t))$ are eliminated by restricting valid terms to (meta-level) β-normal η-long forms (as in [10]), leading to a faithful representation of syntax. Finally, a central feature is that it supports non-annotated syntaxes, allowing to define theories with their true syntax.

Example A dependently-typed λ-calculus (without universes) can be defined by the theory $\mathbb{T}_{\lambda\Pi} := (\Sigma_{\lambda\Pi}, \mathcal{R}_{\lambda\Pi})$ where $\Sigma_{\lambda\Pi}$ is the signature at the left and $\mathcal{R}_{\lambda\Pi}$ is the rewriting system containing only the rewrite rule $@(\lambda(x.\mathtt{t}(x)), \mathtt{u}) \longrightarrow \mathtt{t}(\mathtt{u})$.

$$\begin{array}{rlcrl}
\mathsf{Ty} : & \square & \xrightarrow{|-|} & \mathsf{Ty} :: & \square \\
\mathsf{Tm} : & (\mathtt{A} : \mathsf{Ty}) \to \square & \xrightarrow{|-|} & \mathsf{Tm} :: & (\mathtt{A} :: \mathsf{ty}) \to \square \\
\Pi : & (\mathtt{A} : \mathsf{Ty})(\mathtt{B} : (x : \mathsf{Tm}\ \mathtt{A}) \to \mathsf{Ty}) \to \mathsf{Ty} & \xrightarrow{|-|} & \Pi :: & (\mathtt{A} :: \mathsf{ty})(\mathtt{B} :: (x :: \mathsf{tm}) \to \mathsf{ty}) \to \mathsf{ty} \\
\lambda : & \{\mathtt{A} : \mathsf{Ty}\}\{\mathtt{B} : (x : \mathsf{Tm}\ \mathtt{A}) \to \mathsf{Ty}\} & \xrightarrow{|-|} & \lambda :: & (\mathtt{t} :: (x :: \mathsf{tm}) \to \mathsf{tm}) \to \mathsf{tm} \\
& (\mathtt{t} : (x : \mathsf{Tm}\ \mathtt{A}) \to \mathsf{Tm}\ \mathtt{B}(x)) \to \mathsf{Tm}\ \Pi(\mathtt{A}, x.\mathtt{B}(x)) & & & \\
@ : & \{\mathtt{A} : \mathsf{Ty}\}\{\mathtt{B} : (x : \mathsf{Tm}\ \mathtt{A}) \to \mathsf{Ty}\} & \xrightarrow{|-|} & @ :: & (\mathtt{t} :: \mathsf{tm})(\mathtt{u} :: \mathsf{tm}) \to \mathsf{tm} \\
& (\mathtt{t} : \mathsf{Tm}\ \Pi(\mathtt{A}, x.\mathtt{B}(x)))(\mathtt{u} : \mathsf{Tm}\ \mathtt{A}) \to \mathsf{Tm}\ \mathtt{B}(\mathtt{u}) & & &
\end{array}$$

While the signature at the left is a specification of $\mathbb{T}_{\lambda\Pi}$'s typing rules, the *pre-signature* at the right specifies its syntax. For instance, the entry for λ specifies that it takes an argument of sort tm, where it binds a variable x of sort tm, and produces a term of sort tm.

Signatures and pre-signatures are of course not at all unrelated entities: each signature gives rise to an associated pre-signature through the *dependency erasure* function $|-|$, establishing the link between the typing layer and the raw syntax layer of CompLF theories.

Note that arguments marked with $\{-\}$ are removed by the erasure function. These are called *erased arguments* and enable the definition of theories with non-annotated syntaxes. They are thus absent from the syntax, but still appear in the typing rules (similar to [15]). For instance, when typing $\lambda(x.t)$ one has to provide derivations of the following four premises.

$$\frac{\Gamma \vdash \qquad \Gamma \vdash A : \mathsf{Ty} \qquad \Gamma, x : \mathsf{Tm}\ A \vdash B : \mathsf{Ty} \qquad \Gamma, x : \mathsf{Tm}\ A \vdash t : \mathsf{Tm}\ B}{\Gamma \vdash \lambda(x.t) : \mathsf{Tm}\ \Pi(A, x.B)}$$

If in one hand erased arguments capture the true syntax used in type theories, they make typing a non-trival and in general undecidable task. We now address this issue.

Bidirectional typing algorithms [14, 1, 12, 4, 13] are characterized by featuring two modes: inference ($t \Rightarrow T$) and checking ($t \Leftarrow T$), which allow to specify the flow of information in typing rules. For instance, the following rule explains how to type $\lambda(x.t)$: the terms A and B should not be guessed, but recovered from the type, which should be given as input. Bidirectional typing thus complements erased arguments very well, by explaining how they can be recovered.

$$\frac{C \longrightarrow_{\mathsf{whnf}} \Pi(A, x.B) \qquad \Gamma, x : \mathsf{Tm}\ A \vdash t \Leftarrow \mathsf{Tm}\ B}{\Gamma \vdash \lambda(x.t) \Leftarrow \mathsf{Tm}\ C}$$

2nd contribution We propose a bidirectional typing algorithm for CompLF. Its main distinguishing feature is that it is not designed for a specific theory, but instead is *theory-agnostic*. In order to use it, we first have to give modes to the signature defining the theory in a *mode-correct* way — a condition whose technical definition we do not give here. Each term-level syntactic constructor and non-erased argument is thus marked with either + (infer) or − (check).

If the rewrite system satisfies confluence and subject reduction, the algorithm is sound, and if furthermore it satisfies strong normalization, it is also complete for the *well-moded* terms.

For instance, the entry for λ in $\Sigma_{\lambda\Pi}$ can be annotated as $\lambda^- : \{\mathtt{A} : \mathsf{Ty}\}\{\mathtt{B} : (x : \mathsf{Tm}\ \mathtt{A}) \to \mathsf{Ty}\}(\mathtt{t} : (x : \mathsf{Tm}\ \mathtt{A}) \to \mathsf{Tm}\ \mathtt{B}(x))^- \to \Pi(\mathtt{A}, x.\mathtt{B}(x))$, specifying the bidirectional rule for λ shown previously. Assuming we give the expected modes to the other entries in $\Sigma_{\lambda\Pi}$ (e.g. as in [13]), the well-moded terms are exactly the β-normal forms. But if the user wants abstractions to be inferable, they can instead make $\mathtt{A}$ explicit and take $\lambda^+ : (\mathtt{A} : \mathsf{Ty})^-\{\mathtt{B} : (x : \mathsf{Tm}\ \mathtt{A}) \to \mathsf{Ty}\}(\mathtt{t} : (x : \mathsf{Tm}\ \mathtt{A}) \to \mathsf{Tm}\ \mathtt{B}(x))^+ \to \Pi(\mathtt{A}, x.\mathtt{B}(x))$, in which case all terms are well-moded. Our algorithm thus generalizes other ones which either chose λ^- [14, 1] or λ^+ [12], by supporting both styles.

The algorithm has been implemented and is available at `https://github.com/thiagofelicissimo/complf`. The directory `test` shows some of the theories that can be defined in CompLF and used with our algorithm: MLTT with Taski-style universes, universe polymorphism, HOL, etc.

References

[1] Andreas Abel and Thorsten Altenkirch. A partial type checking algorithm for type: Type. *Electronic Notes in Theoretical Computer Science*, 229(5):3–17, 2011.

[2] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a logical framework based on the $\lambda\pi$-calculus modulo theory. *Manuscript*, 2016.

[3] Frédéric Blanqui, Gilles Dowek, Émilie Grienenberger, Gabriel Hondet, and François Thiré. Some axioms for mathematics. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPIcs*, pages 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[4] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys (CSUR)*, 54(5):1–38, 2021.

[5] Michael Färber. Safe, fast, concurrent proof checking for the lambda-pi calculus modulo rewriting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 225–238, 2022.

[6] Thiago Felicissimo. Adequate and computational encodings in the logical framework dedukti. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel*, volume 228 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[7] Harald Ganzinger, Frank Pfenning, and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In *Automated Deduction—CADE-16: 16th International Conference on Automated Deduction Trento, Italy, July 7–10, 1999 Proceedings 16*, pages 202–206. Springer, 1999.

[8] Robert Harper. An equational logical framework for type theories. *arXiv preprint arXiv:2106.01484*, 2021.

[9] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[10] Robert Harper and Daniel R Licata. Mechanizing metatheory in a logical framework. *Journal of functional programming*, 17(4-5):613–673, 2007.

[11] Gabriel Hondet and Frédéric Blanqui. The new rewriting engine of dedukti (system description). In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 35:1–35:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[12] Meven Lennon-Bertrand. Complete Bidirectional Typing for the Calculus of Inductive Constructions. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[13] Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta informaticae*, 102(2):177–207, 2010.

[14] Conor McBride. Basics of bidirectionalism. `https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionalism/`.

[15] Jason Reed. Redundancy elimination for lf. *Electronic Notes in Theoretical Computer Science*, 199:89–106, 2008.

[16] Taichi Uemura. A general framework for the semantics of type theory. *arXiv preprint arXiv:1904.04097*, 2019.

Revisiting Containers in Cubical Agda

Thorsten Altenkirch[1] and Stefania Damato[1]

[1] University of Nottingham, Nottingham, UK
`{thorsten.altenkirch, stefania.damato}@nottingham.ac.uk`

We present ongoing work on a type-theoretic literature review of the state of the art on containers, as well as a Cubical Agda formalisation of generalised containers.

Strict positivity An *inductive type* X is a type given by a list of constructors, each specifying a way to form an element of X. Defining types inductively is a central notion in Martin-Löf type theory, with examples including the natural numbers $\mathbb{N}$, lists, finite sets, and many more. In this setting, we usually want to be able to make sense of our inductive definitions predicatively, with elements of the type being generated 'in stages'. The condition we would like to impose on our definitions is that they are *strictly positive*. This roughly means that the constructors of X only allow X to appear in input types that are arrows if it appears to the right. So we allow constructors like `c:(ℕ → X) → X`, but not `d:(X → ℕ) → X` or `e:((X → ℕ) → ℕ) → X`. In general, we want to avoid definitions that are not strictly positive, as they can lead to inconsistencies under certain assumptions (like classical logic), so we would like a semantic description of strict positivity in order for our systems to only admit such types. Containers help us do exactly this.

What are containers? A (ordinary) *container* $S \lhd P$ is a set of shapes $S : \mathsf{Set}$ and a family of positions over those shapes $P : S \to \mathsf{Set}$. Every strictly positive type can be thought of as a well-founded tree whose nodes are labelled by elements s of S, and where node s has $P\,s$ many subtrees. E.g. the `List` data type is given as a container by $(n : \mathbb{N}) \lhd (\mathtt{Fin}\,n)$. The shape of a list is a natural number $n : \mathbb{N}$ representing its length, and given a length n, the data of a list is stored at the positions, which are the elements of a finite set of size n, $\mathtt{Fin}\,n$.

To every container $S \lhd P$, we associate a functor $[\![S \lhd P]\!] : \underline{\mathbf{Set}} \to \underline{\mathbf{Set}}$ defined as follows.

- On objects $X : \mathsf{Set}$, we have $[\![S \lhd P]\!]\,X :\equiv \sum(s : S)(P\,s \to X)$.
- On morphisms $f : X \to Y$, we have $[\![S \lhd P]\!]\,f\,(s, g) :\equiv (s, f \circ g)$.

This functor reflects the idea that strictly positive types are simply memory locations in which data can be stored. E.g. the container functor $[\![(n : \mathbb{N}) \lhd (\mathtt{Fin}\,n)]\!]$ allows us to represent concrete lists. The list of `Char`s ['r', 'e', 'd'] is represented as $(3, (0 \mapsto `r'; 1 \mapsto `e'; 2 \mapsto `d') : \sum\ (n : \mathbb{N})\,(\mathtt{Fin}\,n \to \mathtt{Char})$. Containers are also known in the literature as polynomial functors [9, 10]. W-types are the initial algebras of container functors.

Our contribution Over the years, containers have been studied extensively [1, 7, 5]. Some of the key developments on containers are presented using a heavily category-theoretic approach—in particular, they are presented as constructions in the internal language of locally Cartesian closed categories (LCCCs) with disjoint coproducts and W-types (also called Martin-Löf categories). We felt that adapting these results using a more type-theoretic approach would be beneficial for a few reasons. Firstly, using LCCCs to describe models of dependent type theory is too restrictive and not entirely precise (e.g. setoids are not an LCCC [11] but still model dependent type theory). Secondly, we wanted a more accessible presentation of containers for programmers and computer scientists, who might have less of a thorough background in category theory.

To this end, we present ongoing work on a review paper offering a comprehensive and updated type-theoretic view of the state of the art on containers [4]. The paper presents all the established results on (ordinary) containers, discusses other kinds of containers, introduces generalised containers [6], and does so in the language of type theory. To supplement this study, we formalised several results on containers in Cubical Agda [8]. We have two proofs in Cubical Agda of the central result that the container extension functor $[\![\_]\!]$ mapping containers to functors is full and faithful. This was proven for the case of generalised containers, which generalise ordinary containers in that they are parameterised by an arbitrary category $\underline{\mathbf{C}}$ and give rise to functors of type $\underline{\mathbf{C}} \to \underline{\mathbf{Set}}$. One follows the proof given in [1], and the other is a new proof that makes use of the Yoneda lemma. While these two proofs are fully formalised, the review paper as well as a formalisation of additional results is work in progress.

One of the consequences of $[\![\_]\!]$ being full and faithful is that we obtain a characterisation of natural transformations between container functors (i.e. polymorphic functions on strictly positive types) as container morphisms. A container morphism $(S \lhd P) \to (T \lhd Q)$ is a pair $u \colon S \to T$ and $f \colon (s : S) \to Q\,(u\,s) \to P\,s$, e.g. container morphisms between lists are given by a pair $u \colon \mathbb{N} \to \mathbb{N}$ and $f \colon (n : \mathbb{N}) \to \texttt{Fin}\ (u\,n) \to \texttt{Fin}\ n$. This result tells us that *any* polymorphic function on lists (such as tail and reverse) can be represented as such a pair, supporting the claim that containers are a canonical way of representing strictly positive types.

Our formalisation makes use of the category theoretic definitions available in the Cubical library. The Cubical mode of Agda avoids us having to postulate functional extensionality, facilitates the use of heterogenous equality, and allows for future generalisations related to higher inductive types (discussed below).

Future work Our survey of containers was primarily motivated by our current interest in applying them to obtain semantics for quotient inductive-inductive types (QIITs). Our end goal is to provide a canonical way to represent QIIT specifications that admit an initial algebra, i.e. the strictly positive ones. Our approach is to ‘containerify’ the semantics given in [2] to obtain a semantics for strictly positive QIITs. More details on this can be found at [3].

References

[1] M. Abbott, T. Altenkirch, and N. Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005. Applied Semantics: Selected Topics.

[2] T. Altenkirch, P. Capriotti, G. Dijkstra, N. Kraus, and F. Nordvall Forsberg. Quotient inductive-inductive types. In C. Baier and U. Dal Lago, editors, *FoSSACS*, pages 293–310. Springer, 2018.

[3] T. Altenkirch and S. Damato. Specifying QIITS using containers. Abstract submitted to HoTT/UF, 2023. Available at https://stefaniatadama.com/talks/abstract_hott_uf_2023.pdf.

[4] T. Altenkirch and S. Damato. A type-theoretic survey of containers, 2023. In preparation.

[5] T. Altenkirch, N. Ghani, P. Hancock, C. McBride, and P. Morris. Indexed containers. *Journal of Functional Programming*, 25:e5, 2015.

[6] T. Altenkirch and A. Kaposi. A container model of type theory. TYPES abstract, 2021.

[7] T. Altenkirch, P. Levy, and S. Staton. Higher-order containers. In F. Ferreira, B. Löwe, E. Mayordomo, and L. Mendes Gomes, editors, *Programs, Proofs, Processes*, pages 11–20, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[8] Stefania Damato. A formalisation of generalised containers in Cubical Agda. Formalisation, 2023. Available at https://github.com/stefaniatadama/TYPES-23.

[9] N. Gambino and M. Hyland. Wellfounded trees and dependent polynomial functors. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, pages 210–225, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[10] J. Kock. Notes on polynomial functors, October 2009. Available at https://mat.uab.cat/~kock/cat/polynomial.pdf.

[11] N. Kraus and T. Altenkirch. Setoids are not an LCCC. Unpublished note, 2012. Available at https://www.cs.nott.ac.uk/~psznk/docs/setoids.pdf.

Engineering logical relations for MLTT in Coq

Arthur Adjedj[12], Meven Lennon-Bertrand[3], Kenji Maillard[1], and Loïc Pujet[1]

[1] Gallinette Project-Team, Inria, Nantes, France
[2] ENS Paris-Saclay, Gif-sur-Yvette, France
[3] University of Cambridge, UK

Abstract

We report on a mechanization in the Coq proof assistant of the decidability of conversion and type-checking for Martin-Löf Type Theory (MLTT), extending a previous Agda formalization. Our development proves the decidability not only of conversion, but also of type-checking, using bidirectional derivations that are canonical for typing. Moreover, we wish to narrow the gap between the object theory we formalize (currently MLTT with Π, Σ, $\mathbb{N}$ and one universe) and the metatheory used to prove the normalization result, *e.g.,* MLTT, to a mere difference of universe levels. We thus avoid induction-recursion or impredicativity, which are central in previous work. Working in Coq, we also investigate how its features, including universe polymorphism and the metaprogramming facilities provided by tactics, impact the development of the formalization compared to the development style in Agda. The development is freely accessible on GitHub [2].

MLTT and its metatheory Establishing meta-theoretic properties such as the existence of canonical forms or decidability of the derivability of judgements of dependent type systems is a notoriously complex endeavour. For instance, the MetaCoq project [11, 12] aims at entirely formalizing the Predicative Calculus of Cumulative Inductive Constructions (PCUIC) underlying Coq, and showing the correctness of an implementation of a typechecker. However, to do so it assumes an axiom stating that the theory is normalizing.

Indeed, for these meta-theoretical properties, type dependency precludes most proof strategies, which ultimately rely on a stratification between types and terms. To go beyond these limitations, Abel et al. [1] formalize an inductive-recursive [5] definition of a logical relation for a representative fragment of MLTT in Agda, to show normalization and decidability of conversion for this theory. This technique was further extended to more complex theories [6, 10].

The use of the induction-recursion scheme however introduces a new gap between the object theory being formalized (which only supports a handful of simple inductive types), and the meta-theory used to formalize the result. While exploring normalization proofs for complex inductive schema is a very valuable endeavour, we wish to go the other way around, and narrow this gap by using only regular indexed inductive types. This is both a requirement and a benefit of working in Coq, which only handles this class of inductive types. Thus, we reformulate the logical relation using *small induction-recursion*, which can in turn be encoded using simple indexed inductive types [7]. This strategy requires definitions that are replicated across several universe levels, for which the universe polymorphic features of Coq come in handy.

A bidirectional presentation of MLTT In [1], only decidability of conversion is shown. While this is definitely the most intricate part of showing decidability of type-checking, going from the former to the latter is non-trivial. Indeed, type-checking for MLTT as defined in [1] is not, in general, decidable, for lack of annotations [4, 15].

In our development, we show decidability of typing, by extending algorithmic conversion-checking to a full account of algorithmic typing, described in a bidirectional fashion [8, 9].

Following the strategy implemented for instance by Coq, we use annotated (Church-style) abstractions, so that inference is complete, *i.e.,* every well-typed term infers a type.

More generally, ideas from bidirectional typing help greatly in guiding the definition and handling of the algorithmic parts of the system. For instance, for most proofs on the algorithmic system we rely on a custom induction principle which threads the invariants maintained by a bidirectional algorithm, giving us extra hypotheses for each induction step. This lets us handle once and for all these invariants that are required for most proofs, rather than bundling them in the predicate proven by induction, which would mean showing their preservation again during each proof by induction.

Three logical relations in one We show that a bidirectional presentation of our type theory is equivalent to its standard declarative presentation given in the Martin-Löf Logical Framework. To do so, we parametrize the definition of the logical relation by a generic typing interface. The interface is instantiated 3 times: once with the declarative typing, once with a mixed system with declarative typing and algorithmic conversion, and finally with a fully algorithmic system using bidirectional typing. These different instances are used to gradually show properties of the system: the declarative instance lets us show enough good properties of the declarative system to be able to show that the mixed system fits the generic typing interface; the mixed instance proves that declarative and algorithmic conversion coincide, which is used to show that bidirectional typing fits in the generic interface; finally the fully algorithmic instance gets us the desired equivalence, showing that the type-checking algorithm is sound and complete.

Engineering aspects We rely on Autosubst [14, 3] to deal with all the aspect of the raw syntax, defining untyped renamings and substitutions, generating boilerplate lemmas on these, as well as providing tactical support to discharge equational obligations.

To support working with multiple different notions of conversion and typing, sometimes simultaneously, we devised a generic notation system based on type classes in the Math Classes style [13], with a system of tags that lets us disambiguate between the different notions when needed, but can be ignored safely when working on a single notion or parametrically.

To ease the development, we use tactics to provide for judgement-independent notions, e.g. we use a single `irrelevance` tactic to discharge goals requiring a lemma stating that some form of the logical relation is irrelevant in some of its parameters. An important part of the work achieved by the definition of the logical relation consist in its generalization of typing contexts through Kripke-style quantifications over renamings and substitutions, and we use instantiation tactics to automatically apply lemmas to the relevant hypotheses.

Future work We could add more universes to obtain a hierarchy of arbitrary finite length, and see no theoretical obstacle in doing so. We plan to extend the formalization to a scheme of indexed inductive types used in Coq, hence narrowing the gap between the object theory and the metatheory used to prove its normalization: this would lead to a formalization of MLTT with n universes into MLTT with $n+k$ universes for a (small but strictly positive) constant k.

On the bidirectional side, we do not cover the common pattern, used for instance in the kernel of Agda, of having some terms that only check (typically, unannotated abstraction). It would be interesting to obtain a decidability result that covers these as well.

Finally, there is a large space left to improve automation, taking inspiration from the rich Coq ecosystem. Indeed, the main difficulty for a proof by logical relations is in the setup of the relation, but most proof obligations are rather repetitive and unsurprising. While our tactics already relieve us from quite a bit of this tedious work, they are far from making it all disappear.

References

[1] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, 2(POPL):23:1–23:29, 2018.

[2] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, and Loïc Pujet. Logical Relation for MLTT in Coq. `https://github.com/CoqHott/logrel-coq`, 2023.

[3] Adrian Dapprich. Generating Infrastructural Code for Terms with Binders using MetaCoq and OCaml. Bachelor thesis, Saarland University, 2021.

[4] Gilles Dowek. The undecidability of typability in the Lambda-Pi-calculus. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, pages 139–145. Springer Berlin Heidelberg, 1993.

[5] Peter Dybjer and Anton Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 124(1-3):1–47, 2003.

[6] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.

[7] Peter Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. Small Induction Recursion. In Masahito Hasegawa, editor, *Typed Lambda Calculi and Applications*, pages 156–172, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[8] Meven Lennon-Bertrand. Complete Bidirectional Typing for the Calculus of Inductive Constructions. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 24:1–24:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[9] Meven Lennon-Bertrand. *Bidirectional Typing for the Calculus of Inductive Constructions*. PhD thesis, Nantes Université, 2022.

[10] Loïc Pujet and Nicolas Tabareau. Impredicative Observational Equality. *Proc. ACM Program. Lang.*, 7(POPL), 2023.

[11] 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.

[12] 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):8:1–8:28, 2020.

[13] Bas Spitters and Eelis Van Der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.

[14] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 166–180. ACM, 2019.

[15] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, 2006.

A Lock Calculus for Multimode Type Theory

Andreas Nuyts

imec-DistriNet, KU Leuven, Belgium

Abstract

Multimode type theory (MTT) is parametrized by a *mode theory*: a 2-category whose objects, morphisms and 2-cells serve as internal modes, modalities and 2-cells. So far, this mode theory has remained in the metatheory, with syntactic modal type and term formers being indexed by metatheoretic gadgets. Building a syntactic lock calculus on top of the mode theory has several advantages: the modal aspects of substitution take the form of more familiar syntactical substitutions, the lock operation on contexts can be axiomatized as *pseudo*functorial so that models of MTT no longer need to be strict(ified), and we can have internal mode, modality and 2-cell polymorphism with intensional 2-cell equality.

Notation 1. Throughout the abstract, we let p, q, r, s stand for modes, μ, ν, ρ for modalities, α for 2-cells, $\mathfrak{m}, \mathfrak{n}, \mathfrak{o}$ for lock variables, $\mathfrak{s}, \mathfrak{t}, \mathfrak{u}, \mathfrak{v}$ for lock terms, and $\mathfrak{S}, \mathfrak{T}$ for lock substitutions.

Lock variables and lock terms Before we consider what a lock calculus for MTT [GKNB21] should look like, we first modify the original MTT notation so that locks can be referred to via lock variables, which can be substituted with lock terms:[1]

Original	$\Gamma, \text{🔒}_{\mu}$	$\Gamma, \mu \mid x : T$	x_{α}	$\langle \mu \mid T \rangle$	$\mathsf{mod}_{\mu}\, t$
Named	$\Gamma, \mathfrak{m} : \text{🔒}_{\mu}$	$\Gamma, x : \{\mathfrak{m} : \text{🔒}_{\mu}\} T$	$x(\text{🔑}_{\alpha}(\mathfrak{n}))$	$(\mathfrak{m} : \text{🔒}_{\mu}) \to T$	$\lambda(\mathfrak{m} : \text{🔒}_{\mu}).t$

Original	$\mathsf{let}_{\nu}\, (\mathsf{mod}_{\mu}\, x = s)\, \mathsf{in}\, t$
Named	$\mathsf{let}\, (\mathfrak{n} : \text{🔒}_{\nu} \vdash \lambda(\mathfrak{m} : \text{🔒}_{\mu}).x(\mathfrak{n})(\mathfrak{m}) = s)\, \mathsf{in}\, t$

This notation makes several inference rules look familiar or at least reasonable:

$$\frac{\Gamma, \mathfrak{m} : \text{🔒}_{\mu} \vdash T\ \mathsf{type}\ @\ p \qquad \mu : p \to q}{\Gamma, x : \{\mathfrak{m} : \text{🔒}_{\mu}\} T\ \mathsf{ctx}\ @\ q} \qquad \frac{\Gamma, \mathfrak{m} : \text{🔒}_{\mu} \vdash t : T\ @\ p \qquad \mu : p \to q}{\Gamma \vdash \lambda(\mathfrak{m} : \text{🔒}_{\mu}).t : (\mathfrak{m} : \text{🔒}_{\mu}) \to T\ @\ q}$$

$$\frac{\dfrac{\alpha : \mu \Rightarrow \nu : p \to q}{\mathfrak{n} : \text{🔒}_{\nu} \vdash \text{🔑}_{\alpha}(\mathfrak{n}) : \text{🔒}_{\mu}\ @\ q \to p}}{\Gamma, x : \{\mathfrak{m} : \text{🔒}_{\mu}\} T, \mathfrak{n} : \text{🔒}_{\nu} \vdash x(\text{🔑}_{\alpha}(\mathfrak{n})) : T[\text{🔑}_{\alpha}(\mathfrak{n})/\mathfrak{m}]\ @\ p}$$

In the intermediate step of the last rule, we already encounter a novel *lock term* $\text{🔑}_{\alpha}(\mathfrak{n})$.

Lock calculus The lock calculus LC has the following judgement forms, and we give their intuitive and semantical meaning:[2]

$\Psi\ \mathsf{ltele}\ @\ q \to p$	Ψ is a lock telescope $q \to p$	$[\![\Psi]\!]$ is a functor $[\![q]\!] \to [\![p]\!]$.
$\Psi \vdash \mathfrak{t} : \text{🔒}_{\mu}$	$\mathfrak{t}$ is a lock term in ctx. Ψ	$[\![\mathfrak{t}]\!] : [\![\Psi]\!] \to [\![\text{🔒}_{\mu}]\!]$ is a nat. transf.
$\Psi \vdash e : \mathfrak{s} = \mathfrak{t} : \text{🔒}_{\mu}\ @\ q \to p$	e proves intensional equality	$[\![\mathfrak{s}]\!] = [\![\mathfrak{t}]\!]$.
$\Psi \vdash \mathfrak{T} : \Phi\ @\ q \to p$	$\mathfrak{T}$ is a lock subst. from Ψ to Φ	$[\![\mathfrak{T}]\!] : [\![\Psi]\!] \to [\![\Phi]\!]$ is a nat. transf.
$\Psi \vdash e : \mathfrak{S} = \mathfrak{T} : \Phi\ @\ q \to p$	e proves intensional equality	$[\![\mathfrak{S}]\!] = [\![\mathfrak{T}]\!]$.

The origin of locks and lock terms remains the external mode theory:

$$\frac{}{()\ \mathsf{ltele}} \qquad \frac{\Psi\ \mathsf{ltele}\ @\ r \to q \qquad \mu : p \to q}{\Psi, \mathfrak{m} : \text{🔒}_{\mu}\ \mathsf{ltele}\ @\ r \to p}$$

[1] This is an improvement of the tick notation proposed in [Nuy20, ch. 5.3].

[2] In the style of generalized algebraic theories [Car86, Car78], we omit judgments for definitional equality.

$$\frac{\mu : p \to q}{\mathfrak{m} : \text{🔒}_{\mu} \vdash \mathfrak{m} : \text{🔒}_{\mu} \mathbin{@} q \to p} \qquad \frac{\alpha : \mu \Rightarrow \nu : p \to q \qquad \Psi \vdash \mathfrak{t} : \text{🔒}_{\nu}}{\Psi \vdash \text{🔑}_{\alpha}(\mathfrak{t}) : \text{🔒}_{\mu} \mathbin{@} q \to p}.$$

Lock substitutions arise from terms and compose horizontally. Vertical composition of substitution is a special case of general substitution, which actually works by syntactical substitution of lock terms for lock variables, and should extend to MTT:

$$\frac{\Psi \vdash \mathfrak{t} : \text{🔒}_{\mu}}{\Psi \vdash (\mathfrak{t}/\mathfrak{m}) : (\mathfrak{m} : \text{🔒}_{\mu})} \qquad \frac{\Psi' \vdash \mathfrak{S} : \Phi' \mathbin{@} r \to q \qquad \Psi \vdash \mathfrak{T} : \Phi \mathbin{@} q \to p}{\Psi', \Psi \vdash \mathfrak{S}, \mathfrak{T} : \Phi', \Phi \mathbin{@} r \to p} \qquad \frac{\Psi \vdash J \qquad \Phi \vdash \mathfrak{S} : \Psi}{\Phi \vdash J[\mathfrak{S}]}$$

Identity and composite locks MTT originally has strict equality rules $(\Gamma, \text{🔒}_{\mathrm{id}}) = \Gamma$ and $(\Gamma, \text{🔒}_{\mu \circ \nu}) = (\Gamma, \text{🔒}_{\mu}, \text{🔒}_{\nu})$. We wish to turn these into natural isomorphisms, and we start by doing so in the lock calculus. The lock calculus serves to be the internal language of the mode theory, which can be any 2-category. As 2-categories are the horizontal categorification (a.k.a. oidification) [nLa23] of (non-symmetric) monoidal categories, we can draw inspiration from existing calculi for those [JM10, §2.1][Shu16, §2.4.2]. We get constructors for identity and composite locks:

$$\frac{}{\vdash () : \text{🔒}_{\mathrm{id}} \mathbin{@} p \to p} \qquad \frac{\Phi \vdash \mathfrak{s} : \text{🔒}_{\nu} \mathbin{@} r \to q \qquad \Psi \vdash \mathfrak{t} : \text{🔒}_{\mu} \mathbin{@} q \to p}{\Phi, \Psi \vdash (\mathfrak{s}, \mathfrak{t}) : \text{🔒}_{\nu \circ \mu} \mathbin{@} r \to p}$$

These are eliminated using a let-expression. However, in order to be able to intensionally prove naturality of this let-expression w.r.t. its target lock, we will also provide the let expression for intensional equality. In order to be able to split composite and remove identity locks in MTT without the context equations given above, we will even provide the let-expression for MTT terms:[3]

$$\frac{\Phi, \mathfrak{n} : \text{🔒}_{\nu}, \mathfrak{m} : \text{🔒}_{\mu}, \Psi \vdash \mathfrak{u} : \text{🔒}_{\rho} \mathbin{@} s \to o \qquad \Xi \vdash \mathfrak{t} : \text{🔒}_{\nu \circ \mu} \mathbin{@} r \to p}{\Phi, \Xi, \Psi \vdash \mathsf{let}\,((\mathfrak{n}, \mathfrak{m}) = \mathfrak{t})\,\mathsf{in}\,\mathfrak{u} : \text{🔒}_{\rho} \mathbin{@} s \to o}$$

$$\frac{\begin{array}{c}\Phi, \mathfrak{o} : \text{🔒}_{\nu \circ \mu}, \Psi \vdash \mathfrak{u}, \mathfrak{v} : \text{🔒}_{\rho} \mathbin{@} s \to o \qquad \Xi \vdash \mathfrak{t} : \text{🔒}_{\nu \circ \mu} \mathbin{@} r \to p \\ \Phi, \mathfrak{n} : \text{🔒}_{\nu}, \mathfrak{m} : \text{🔒}_{\mu}, \Psi \vdash e : \mathfrak{u}[(\mathfrak{n}, \mathfrak{m})/\mathfrak{o}] = \mathfrak{v}[(\mathfrak{n}, \mathfrak{m})/\mathfrak{o}] : \text{🔒}_{\rho} \mathbin{@} s \to o\end{array}}{\Phi, \Xi, \Psi \vdash \mathsf{let}\,((\mathfrak{n}, \mathfrak{m}) = \mathfrak{t})\,\mathsf{in}\,e : \mathfrak{u}[\mathfrak{t}/\mathfrak{o}] = \mathfrak{v}[\mathfrak{t}/\mathfrak{o}] : \text{🔒}_{\rho} \mathbin{@} s \to o}$$

$$\frac{\begin{array}{c}\Gamma, \mathfrak{o} : \text{🔒}_{\nu \circ \mu}, \Psi \vdash A\,\mathsf{type} \mathbin{@} o \qquad \Xi \vdash \mathfrak{t} : \text{🔒}_{\nu \circ \mu} \mathbin{@} r \to p \\ \Gamma, \mathfrak{n} : \text{🔒}_{\nu}, \mathfrak{m} : \text{🔒}_{\mu}, \Psi \vdash a : A[(\mathfrak{n}, \mathfrak{m})/\mathfrak{o}] \mathbin{@} o\end{array}}{\Gamma, \Xi, \Psi \vdash \mathsf{let}\,((\mathfrak{n}, \mathfrak{m}) = \mathfrak{t})\,\mathsf{in}\,a : A[\mathfrak{t}/\mathfrak{o}] \mathbin{@} o}$$

These let-expressions β-reduce definitionally when $\mathfrak{t}$ is actually a pair. There are similar rules for 🔒_{id}. We can now admit pseudofunctorial models by interpreting the introduction and elimination rules for 🔒_{id} and $\text{🔒}_{\nu \circ \mu}$ via the unitors and compositors of the model.

The metamode We have not yet delivered on our promise to allow internal mode, modality and 2-cell polymorphism, nor have we explained how to use intensional 2-cell equality. In each of these cases, we need to step outside the mode theory to reason about it, and we need a type system for that, so that we can quantify and use a J-rule. To this end, we include another copy of MLTT – to be modelled in a category of sufficiently big sets – called the *metamode* (whose type assignment will be denoted with $a :: A$), and prefix every MTT and LC judgement with a metamode context Θ. (Dependent) types of modes, modalities and 2-cells exist in the metamode and their behaviour depends on the choice of mode theory. When MTT or LC requires a mode, modality or 2-cell, we can take a metamode term in context Θ. Additionally, there will be metamode types of lock and MTT terms, lock substitutions and intensional LC equality (with J-rule):[4]

$$\frac{\Theta \mid \cdot \vdash t : T \mathbin{@} p}{\Theta \vdash \ulcorner t \urcorner :: \mathsf{Tm}_p(T)} \qquad \frac{\Theta \mid \Psi \vdash \mathfrak{t} : \text{🔒}_{\mu} \mathbin{@} q \to p}{\Theta \vdash \ulcorner \mathfrak{t} \urcorner :: \mathsf{LTm}_{q \to p}(\Psi, \mu)} \qquad \frac{\Theta \mid \Psi \vdash e : \mathfrak{s} = \mathfrak{t} : \text{🔒}_{\mu} \mathbin{@} q \to p}{\Theta \vdash \ulcorner e \urcorner :: \mathsf{LTE}_{q \to p}(\Psi, \mu, \mathfrak{s}, \mathfrak{t})}$$

$$\frac{\Theta \vdash t :: \mathsf{Tm}_p(T)}{\Theta \mid \Gamma \vdash \llcorner t \lrcorner : T \mathbin{@} p} \qquad \frac{\Theta \vdash t :: \mathsf{LTm}_{q \to p}(\Psi, \mu)}{\Theta \mid \Psi \vdash \llcorner t \lrcorner : \text{🔒}_{\mu} \mathbin{@} q \to p} \qquad \frac{\Theta \vdash e : \mathsf{LTE}_{q \to p}(\Psi, \mu, \mathfrak{s}, \mathfrak{t})}{\Theta \mid \Psi \vdash \llcorner e \lrcorner : \mathfrak{s} = \mathfrak{t} : \text{🔒}_{\mu} \mathbin{@} q \to p}$$

[3]In MTT, the telescope to the right of Ξ can be quantified over, so we may assume it to be empty or more generally a lock telescope.

[4]In MTT, the entire context can be quantified over, so we may assume it to be empty.

Acknowledgements Thanks to Jon Sterling and Zhixuan Yang for pointing to the right sources [JM10, Shu16], and to Joris Ceulemans and Dominique Devriese for proof-reading. Andreas Nuyts holds a Postdoctoral Fellowship from the Research Foundation - Flanders (FWO). This research is partially funded by the Research Fund KU Leuven.

References

[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.

[Car78] John Cartmell. *Generalised Algebraic Theories and Contextual Categories*. PhD thesis, 1978.

[Car86] John Cartmell. Generalised algebraic theories and contextual categories. *Ann. Pure Appl. Logic*, 32:209–243, 1986. doi:10.1016/0168-0072(86)90053-9.

[GKNB21] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal dependent type theory. *Log. Methods Comput. Sci.*, 17(3), 2021. doi:10.46298/lmcs-17(3:11)2021.

[JM10] Mauro Jaskelioff and Eugenio Moggi. Monad transformers as monoid transformers. *Theoretical Computer Science*, 411(51):4441–4466, 2010. European Symposium on Programming 2009. URL: https://www.sciencedirect.com/science/article/pii/S0304397510004901, doi:https://doi.org/10.1016/j.tcs.2010.09.011.

[nLa23] nLab authors. horizontal categorification. https://ncatlab.org/nlab/show/horizontal+categorification, March 2023. Revision 31.

[Nuy20] Andreas Nuyts. *Contributions to Multimode and Presheaf Type Theory*. PhD thesis, KU Leuven, Belgium, 8 2020. URL: https://anuyts.github.io/files/phd.pdf.

[Shu16] Michael Shulman. Categorical logic from a categorical point of view, July 2016. URL: https://mikeshulman.github.io/catlog/catlog.pdf.

6

Session 7: Foundations of type theory and constructive mathematics

On epimorphisms and acyclic types in univalent type theory

Ulrik Buchholtz[1], Tom de Jong[1], and Egbert Rijke[2]

[1] School of Computer Science, University of Nottingham
[2] Department of Mathematics, University of Ljubljana

We study epimorphisms and acyclic types in univalent mathematics. The epimorphisms are of course a natural object of study, the definition being that a map $f : A \to B$ is an *epimorphism* if for every type X, the precomposition map

$$(B \to X) \xrightarrow{(-)\circ f} (A \to X)$$

is an embedding. Put differently, extensions of maps from A through B are unique if they exist.

The epimorphisms behave quite differently in higher types than do the surjections of sets, as can be seen from considering the higher inductive type of the circle $\mathbb{S}^1$. Recall that this type has a basepoint base : $\mathbb{S}^1$ such that the type of loops base $=_{\mathbb{S}^1}$ base is equivalent to the type of integers $\mathbb{Z}$. While the unique map from the two-element type $\mathbf{2}$ to the unit type $\mathbf{1}$ is a surjection, it is not an epimorphism of types, because one can show that the type of dashed extensions in the diagram

$$\begin{tikzcd} \mathbf{2} \arrow[r] \arrow[d, "{[\mathrm{base},\mathrm{base}]}"'] & \mathbf{1} \arrow[dl, dashed] \\ \mathbb{S}^1 & \end{tikzcd}$$

does not have at most one element, so that the extension is not unique.

Our aim In the classical theory of spaces, it is known that epimorphisms are related to so-called acyclic spaces. We show that the same is true in univalent type theory. Thus, we turn to algebraic topology to answer a question about types, namely: what are the epimorphisms in univalent type theory? In doing so, we contribute to the field of synthetic homotopy theory, where the language of univalent type theory is used to develop algebraic topology without reference to set-based presentations such as topological spaces or simplicial sets.

Acyclic types Classically, a space is acyclic if its reduced integral homology vanishes. This characterization only works assuming Whitehead's principle, and the correct definition in univalent mathematics is the following:

Definition 1. A type A is *acyclic* if its suspension is contractible. We extend this definition to maps by declaring a map $f : A \to B$ to be acyclic if all of its fibers $\mathrm{fib}_f(b) :\equiv \sum_{a:A} f(a) = b$ are.

We recall that the suspension of a type A is the higher inductive type generated by two points N (north) and S (south) and a path from N to S for every element $a : A$.

We can then characterize the epimorphisms as the acyclic maps:

Theorem 2. *A map is an epimorphism if and only if it is acyclic.*

It follows from the theorem that being an epimorphism is a fiberwise notion and therefore we can for example immediately deduce that the epimorphisms are closed under retracts and stable under pullback along arbitrary maps. Moreover, we can show that the epimorphisms are pushout stable and satisfy a 3-for-2 property.

In other words, the class of acyclic maps (equivalently, epimorphisms) enjoys reasonable closure properties. However, it turns out to be somewhat difficult to construct acyclic types. Indeed, the construction of a nontrivial acyclic type answers a question left open in [CR22, Ex. 6.6]. The construction of such a nontrivial acyclic type is our next main contribution.

That such types cannot be sets is because of the following:

Theorem 3. *A set is acyclic if and only if it is contractible.*

Interestingly, to prove this, we need the fact that the generators map $\eta : A \to \mathrm{F}_A$, from A to the free group on A is an injection [MRR88, Chapter X]. This has previously been proved in homotopy type theory [BCDE21] with a recent synthetic proof in [Wär23].

Classically, it's well known that the Higman group [Hig51], given by the presentation

$$H = \langle\, a, b, c, d \mid a = [d, a], b = [a, b], c = [b, c], d = [c, d] \,\rangle,$$

where $[x, y] = xyx^{-1}y^{-1}$ denotes the commutator of x and y, is acyclic (i.e., has an acyclic classifying space), and moreover, this presentation is *aspheric*, meaning that the presentation complex is already a 1-type [DV73]. The presentation complex is easily imported into homotopy type theory as the higher inductive type $\mathrm{B}H$ with a point constructor $\mathrm{pt} : \mathrm{B}H$, four path constructors $a, b, c, d : \Omega\mathrm{B}H$, and four 2-cell constructors corresponding to the relations.

The Eckmann–Hilton argument [Uni13, Theorem 2.1.6] tells us that the higher homotopy groups of any type are abelian. Together with a higher inductive description of the suspension of $\mathrm{B}H$, we can show:

Theorem 4. *The type* $\mathrm{B}H$ *is acyclic.*

It is also possible to construct $\mathrm{B}H$ as a series of iterated pushouts. This description and recent results of David Wärn [Wär23] allow us to prove:

Theorem 5. *The type* $\mathrm{B}H$ *is a 1-type (i.e. its identity types are sets) and it is nontrivial.*

Specifically, we can show that the generators a, b, c and d all have infinite order in the loop space of $\mathrm{B}H$. It is noteworthy that this type-theoretic proof completely avoids classical combinatorial group theory.

Future directions Classically, the acyclic maps form the left class of an accessible orthogonal factorization system, whose right class are the *hypoabelian* maps (i.e., whose π_1-kernels have no non-trivial perfect subgroups). This factorization system exists in all $(\infty, 1)$-toposes [Hoy19], and can be used to derive the McDuff–Segal completion theorem, but with arguments that don't lend themselves to direct internalization in homotopy type theory.

We believe that it is possible to construct the factorization system using Quillen's *plus construction* in HoTT, although, at present, we need to assume additional axioms in the form of Sets Cover, Whitehead's Principle and Countable Choice.

We would also like to investigate whether other classically known acyclic types and maps can be shown to be so in homotopy type theory, such as $\mathrm{B}\,\mathrm{Aut}(\mathbb{N})$ [dlHM83] and $\mathrm{B}\Sigma_\infty \to (\Omega^\infty\Sigma^\infty\mathbb{S}^0)_0$, the Barratt–Priddy(–Quillen) theorem [BP72].

References

[BCDE21] Marc Bezem, Thierry Coquand, Peter Dybjer and Martín Escardó. 'Free groups in HoTT/UF in Agda'. https://www.cs.bham.ac.uk/~mhe/TypeTopology/FreeGroup.html. 2021.

[BP72] Michael Barratt and Stewart Priddy. 'On the homology of non-connected monoids and their associated groups'. In: *Comment. Math. Helv.* 47 (1972), pp. 1–14. DOI: 10.1007/BF02566785.

[CR22] J. Daniel Christensen and Egbert Rijke. 'Characterizations of modalities and lex modalities'. In: *J. Pure Appl. Algebra* 226.3 (2022), Paper No. 106848, 21. DOI: 10.1016/j.jpaa.2021.106848.

[dlHM83] Pierre de la Harpe and Dusa McDuff. 'Acyclic groups of automorphisms'. In: *Comment. Math. Helv.* 58.1 (1983), pp. 48–71. DOI: 10.1007/BF02564624.

[DV73] Eldon Dyer and A. T. Vasquez. 'Some small aspherical spaces'. In: *Journal of the Australian Mathematical Society* 16.3 (1973), pp. 332–352. DOI: 10.1017/S1446788700015147.

[Hig51] Graham Higman. 'A finitely generated infinite simple group'. In: *J. London Math. Soc.* 26 (1951), pp. 61–64. DOI: 10.1112/jlms/s1-26.1.61.

[Hoy19] Marc Hoyois. 'On Quillen's plus construction'. 2019. URL: https://hoyois.app.uni-regensburg.de/papers/acyclic.pdf.

[MRR88] Ray Mines, Fred Richman and Wim Ruitenburg. *A Course in Constructive Algebra*. Universitext. Springer, 1988. DOI: 10.1007/978-1-4419-8640-5.

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

[Wär23] David Wärn. *Path spaces of pushouts*. Preprint. 2023. URL: https://dwarn.se/po-paths.pdf.

The ordinals in set theory and type theory are the same

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

[1] University of Nottingham, Nottingham, UK
{tom.dejong, nicolai.kraus}@nottingham.ac.uk
[2] University of Strathclyde, Glasgow, UK
fredrik.nordvall-forsberg@strath.ac.uk
[3] SonarSource GmbH, Bochum, Germany
chuangjie.xu@sonarsource.com

Set theory or type theory; which one is "better" for constructive mathematics? While we do not dare to offer an answer to this question, we can at least report that when it comes to constructive ordinal theory, the choice between these two foundations is insignificant: the set-theoretic and type-theoretic ordinals coincide. We consider this an interesting finding since ordinals are fundamental in the foundations of set theory and are used in theoretical computer science in termination arguments [7] and semantics of inductive definitions [1, 5].

Comparing ordinals in set theory and (homotopy) type theory In constructive set theory [3], following Powell's seminal work [9], the standard definition of an ordinal is that of a *transitive* set whose elements are again transitive sets. A set x is transitive if for every $y \in x$ and $z \in y$, we have $z \in x$. Note how this definition makes essential use of how the membership predicate $\in$ in set theory is global, by referring to both $z \in y$ and $z \in x$. In type theory, on the other hand, the statement "if $y : x$ and $z : y$ then $z : x$" is ill-formed, and so ordinals need to be defined differently. In the homotopy type theory book [10], an ordinal is defined to be a type equipped with a proposition-valued order relation that is transitive, extensional, and wellfounded [10, §10.3]. Extensionality implies that the underlying type of an ordinal is a set [6].

A priori, the set-theoretic and the type-theoretic approaches to ordinals are thus quite different. One way to compare them is to interpret one foundation into the other. Aczel [2] gave an interpretation of Constructive ZF set theory into type theory using setoids, which was later refined using a higher inductive type $\mathbb{V}$ [10, §10.5], referred to as the *cumulative hierarchy*. Using the set membership relation $\in$ on the cumulative hierarchy, we can construct the subtype $\mathbb{V}_{\mathrm{ord}}$ of elements of $\mathbb{V}$ that are set-theoretic ordinals. Similarly, we write Ord for the type of all type-theoretic ordinals, i.e., for the type of transitive, extensional, and wellfounded order relations. A fundamental result about type-theoretic ordinals is that, using univalence, the type Ord of (small) ordinals is itself a type-theoretic ordinal when ordered by inclusion of strictly smaller initial segments (also referred to as *bounded simulations*), and we show that the type $\mathbb{V}_{\mathrm{ord}}$ of set-theoretic ordinals also canonically carries the structure of a type-theoretic ordinal.

Next, we show that $\mathbb{V}_{\mathrm{ord}}$ and Ord are equivalent, meaning that we can translate between type-theoretic and set-theoretic ordinals. Furthermore, the isomorphism that we construct respects the order structure of $\mathbb{V}_{\mathrm{ord}}$ and Ord, which means that Ord and $\mathbb{V}_{\mathrm{ord}}$ are isomorphic as (large) ordinals. Thus, the set-theoretic and type-theoretic approaches to ordinals coincide in homotopy type theory:

Theorem 1 ([4, Theorem 33] ✿). *The ordinals* Ord *and* $\mathbb{V}_{\mathrm{ord}}$ *are isomorphic (as type-theoretic ordinals). Hence, by univalence, they are equal.* □

Generalising from ordinals to sets Since the subtype $\mathbb{V}_{\mathrm{ord}}$ of $\mathbb{V}$ is isomorphic to Ord, a type of ordered structures, it is natural to ask if there is a type of ordered structures that captures *all* of $\mathbb{V}$. That is, we look for a type T of ordered structures such that Diagram 1 below commutes.

Since $\mathbb{V}$ is $\mathbb{V}_{\text{ord}}$ with transitivity dropped, it is tempting to try to choose T to be Ord without transitivity, i.e., the type of extensional and wellfounded relations. However such an attempt cannot work for cardinality reasons: for example, the set-theoretic ordinal $2 = \{\emptyset, \{\emptyset\}\}$ corresponds to the type-theoretic ordinal α with elements $0 < 1$, but there are more subsets of 2 than subrelations of α. Instead we need additional structure to capture the elements of elements (of elements ...) of sets. To this end, we introduce the type $\mathsf{MEWO}_{\mathsf{cov}}$ of *(covered) marked extensional wellfounded order relations (mewos)*, i.e., extensional wellfounded relations with additional structure in the form of a *marking*.[1] The idea is that the carrier of the order also contains "deeper" elements of elements of the set, with the marking designating the "top-level" elements. Such a marking is *covering* if any element can be reached from a marked top-level element, i.e., if the order contains no "junk". Since every ordinal can be equipped with the trivial covering by marking all elements, the type Ord of ordinals is a subtype of the type of covered mewos. Taking $T = \mathsf{MEWO}_{\mathsf{cov}}$, this gives the inclusion $\mathsf{Ord} \hookrightarrow T$ in Diagram 1.

$$\begin{tikzcd}
\mathbb{V}_{\text{ord}} \arrow[r, "\simeq"] \arrow[d, hook] & \mathsf{Ord} \arrow[d, hook] \\
\mathbb{V} \arrow[r, "\simeq"] & T
\end{tikzcd} \tag{1}$$

To show also $\mathbb{V} \simeq \mathsf{MEWO}_{\mathsf{cov}}$, we develop the theory of covered mewos: the type of covered mewos is itself a covered mewo, with order $<$ given by an appropriately modified notion of bounded simulation (to take the lack of transitivity into account), and covered mewos are closed under both singletons and least upper bounds of arbitrary (small) families of covered mewos. We can then show that indeed $T = \mathsf{MEWO}_{\mathsf{cov}}$ fulfils the requirements of Diagram 1:

Theorem 2 ([4, Theorem 76] ✿). *The structures* $(\mathbb{V}, \in)$ *and* $(\mathsf{MEWO}_{\mathsf{cov}}, <)$ *are equal as covered mewos.* □

Full Paper and Formalisation More details are available in our paper at LICS this year [4]. We have also formalised all our results in Agda. An HTML rendering can be found at the URL https://tdejong.com/agda-html/st-tt-ordinals/index.html.

References

[1] Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 739–782. North-Holland Publishing Company, 1977.

[2] Peter Aczel. The type theoretic interpretation of constructive set theory. In A. MacIntyre, L. Pacholski, and J. Paris, editors, *Logic Colloquium '77*, volume 96 of *Studies in Logic and the Foundations of Mathematics*, pages 55–66. North-Holland Publishing Company, 1978.

[3] Peter Aczel and Michael Rathjen. Notes on constructive set theory. Book draft, available at: https://www1.maths.leeds.ac.uk/~rathjen/book.pdf, 2010.

[4] Tom de Jong, Nicolai Kraus, Fredrik Nordvall Forsberg, and Chuangjie Xu. Set-theoretic and type-theoretic ordinals coincide. In *Proceedings of the 38th Annual ACM/IEEE Symposium on Logic in Computer Science*, 2023. Preprint available at arXiv:2301.10696.

[5] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer, 1999.

[6] Martín Hötzel Escardó et al. Ordinals in univalent type theory in Agda notation. Agda development, HTML rendering available at: https://www.cs.bham.ac.uk/~mhe/TypeTopology/Ordinals.index.html, 2018.

[1]Similar ideas were used by Osius [8] to give a categorical account of set theory.

[7] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.

[8] Gerhard Osius. Categorical set theory: A characterization of the category of sets. *Journal of Pure and Applied Algebra*, 4(1):79–119, 1974.

[9] William C. Powell. Extending Gödel's negative interpretation to ZF. *The Journal of Symbolic Logic*, 40(2):221–229, 1975.

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

Diller-Nahm Bar Recursion

Valentin Blot

Inria, Laboratoire Méthodes Formelles, Université Paris-Saclay

Abstract

We present a generalization of Spector's bar recursion to the Diller-Nahm variant of Gödel's Dialectica interpretation. This generalized bar recursion collects witnesses of universal formulas in sets of approximation sequences to provide an interpretation to the double-negation shift principle. The interpretation is presented in a fully computational way, implementing sets via lists. We also present a demand-driven version of this extended bar recursion manipulating partial sequences rather than initial segments. We explain why in a Diller-Nahm context there seems to be several versions of this demand-driven bar recursion, but no canonical one.

Gödel's functional interpretation [4], also known as the Dialectica interpretation (from the name of the journal it was published in) is a translation from intuitionistic arithmetic into the Σ^0_2 fragment of intuitionistic arithmetic in finite types. If π is a proof of arithmetical formula A, then the functional interpretation of π is a proof of a formula $A^D \equiv \exists\vec{x}\forall\vec{y}A_D\left(\vec{x},\vec{y}\right)$ where A_D is quantifier-free. This formula can be understood as asserting that some two-player game has a winning strategy: there exists a strategy $\vec{x}$ such that for all strategy $\vec{y}$, $\vec{x}$ wins against $\vec{y}$, that is, $A_D\left(\vec{x},\vec{y}\right)$ holds. By the witness property, the proof of A^D yields a proof of $\forall\vec{y}A_D\left(\vec{t},\vec{y}\right)$ for some sequence of terms $\vec{t}$ in system T: simply-typed λ-calculus with recursion over natural numbers at all finite types. This sequence $\vec{t}$ of programs is the computational content of π under the Dialectica interpretation.

Since the negative translation of every axiom of arithmetic is provable in intuitionistic arithmetic, the Dialectica interpretation combined with a negative translation provides an interpretation of classical arithmetic. When it comes to classical analysis (classical arithmetic plus the axiom of countable choice) this is not true anymore, as the negative translation of the axiom of choice fails to be an intuitionistic consequence of the axiom of choice. Spector's bar recursion operator [8] provides a Dialectica interpretation of the double-negation shift (DNS) principle, from which one can derive intuitionistically any formula from its negative translation. Applying this to the axiom of countable choice, Spector obtains an interpretation of classical analysis.

Interpreting the contraction rule $A \Rightarrow A \wedge A$ in Gödel's original interpretation requires (besides the $\lambda x.\langle x, x\rangle$ component) a program that, given a witness M and two potential counterwitnesses x and y of A such that either x or y wins against M, answers with a single counterwitness that wins against M. Doing that relies on the decidability of winningness, which ultimately relies on the decidability of atomic formulas of the source logic (which is true in arithmetic). In order to get rid of this decidability requirement, Diller and Nahm [2] defined a variant of Gödel's interpretation where the programs provide a finite set of counterwitnesses, with the requirement that at least one is correct. In the previous example, the program interpreting the contraction rule answers with the set $\{x; y\}$ and does not have to decide which one is correct. The first contribution of this paper is the extension of Spector's bar recursion to the Diller-Nahm setting. Our operator has a lot in common with the extension of bar recursion to the Herbrand functional interpretation of non-standard arithmetic [3], though there are notable differences which are discussed.

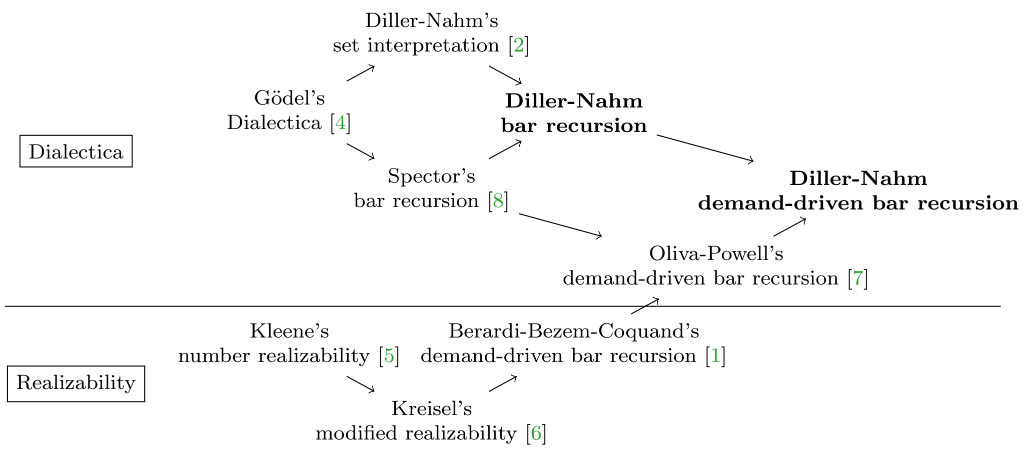

Diller-Nahm's set interpretation [2]
Gödel's Dialectica [4]
Diller-Nahm bar recursion
Dialectica
Spector's bar recursion [8]
Diller-Nahm demand-driven bar recursion
Oliva-Powell's demand-driven bar recursion [7]
Kleene's number realizability [5]
Berardi-Bezem-Coquand's demand-driven bar recursion [1]
Realizability
Kreisel's modified realizability [6]

Figure 1: Contributions of the paper, **in bold font**

Berardi, Bezem and Coquand [1] adapted Spector's bar recursion from Gödel's Dialectica to Kreisel's modified realizability [6]. Their operator also behaves differently from Spector's original bar recursion as it is demand-driven: it computes the choice sequence in an order that is driven by the environment, rather than in the natural order on natural numbers. This provides a more natural computational interpretation to the axiom of countable choice. More recently, Oliva and Powell [7] adapted Berardi-Bezem-Coquand's operator to Gödel's Dialecica interpretation and obtained a demand-driven bar-recursive interpretation of the axiom of countable choice in this setting. The second contribution of this paper is the definition of a demand-driven bar recursion operator in the Diller-Nahm setting.

Figure 1 summarizes the two contributions of this paper as well as their relationship to the state of the art. An arrow from X to Y means that Y is an extension/refinement/variant of X, and we distinguish elements that take place in the framework of realizability from those that take place in the framework of Dialectica-style functional interpretations.

This work has been selected for presentation at the FSCD 2023 conference.

References

[1] Stefano Berardi, Marc Bezem, and Thierry Coquand. On the Computational Content of the Axiom of Choice. *Journal of Symbolic Logic*, 63(2):600–622, 1998.

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

[3] Martín Escardó and Paulo Oliva. The herbrand functional interpretation of the double negation shift. *Journal of Symbolic Logic*, 82(2):590–607, 2017.

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

[5] Stephen Cole Kleene. On the Interpretation of Intuitionistic Number Theory. *Journal of Symbolic Logic*, 10(4):109–124, 1945.

[6] Georg Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In *Constructivity in mathematics: Proceedings of the colloquium held at Amsterdam, 1957*, Studies in

Logic and the Foundations of Mathematics, pages 101–128. North-Holland Publishing Company, 1959.

[7] Paulo Oliva and Thomas Powell. Bar recursion over finite partial functions. *Annals of Pure and Applied Logic*, 168(5):887–921, 2017.

[8] 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.

Markov's Principles in Constructive Type Theory

Yannick Forster[1], Dominik Kirst[2,3], Bruno da Rocha Paiva[4], and Vincent Rahli[4]

[1]Inria, Nantes, France, [2]Saarland University, Germany, [3]Ben-Gurion University, Israel, [4]University of Birmingham, UK

Abstract

Markov's principle (MP) is an axiom in some varieties of constructive mathematics, stating that Σ^0_1 propositions (i.e. existential quantification over a decidable predicate on $\mathbb{N}$) are stable under double negation. However, there are various non-equivalent definitions of decidable predicates and thus Σ^0_1 in constructive foundations, leading to non-equivalent Markov's principles. This fact is often overlooked and leads to confusion: At the time of writing, both Wikipedia and nlab claim propositions to be equivalent to MP, which are however only respectively equivalent to two non-equivalent forms of MP.

We give three variants of MP in constructive type theory, along with respective equivalence proofs to different formulations of Post's theorem ("Σ^0_1-predicates with complement in Σ^0_1 are decidable"), stability of termination of computations, the statement that an extended natural number is finite if it is not infinite, and to completeness of natural deduction w.r.t. Tarski semantics over the $(\forall, \to, \bot)$-fragment of classical first-order logic for Σ^0_1-theories. The first definition ($\mathsf{MP}_\mathbb{P}$) uses a purely logical definition of Σ^0_1 for predicates $\mathbb{N} \to \mathbb{P}$, while the second one ($\mathsf{MP}_\mathbb{B}$) relies on type-theoretic functions $\mathbb{N} \to \mathbb{B}$, and the third one ($\mathsf{MP}_\mathsf{PR}$) on a model of computation.

We conclude with the – to the best of our knowledge – first proof that $\mathsf{MP}_\mathbb{B}$ is *not* equivalent to MP_PR using a model via Cohen and Rahli's $\mathrm{TT}^\square_\mathcal{C}$, and pose the open question how to separate $\mathsf{MP}_\mathbb{P}$ from $\mathsf{MP}_\mathbb{B}$ – where the model would have to invalidate unique choice.

Definitions We work in constructive type theory with a universe of propositions $\mathbb{P}$, e.g. in the calculus of inductive constructions (CIC). We define three variants of Markov's principle:

$$\begin{aligned}
\mathsf{MP}_\mathbb{P} &:= \forall A : \mathbb{N} \to \mathbb{P}.\ (\forall n.\ An \lor \neg An) \to && \neg\neg(\exists n.\ An) \to (\exists n.\ An)\\
\mathsf{MP}_\mathbb{B} &:= \forall f : \mathbb{N} \to \mathbb{B}. && \neg\neg(\exists n.\ fn = \mathsf{true}) \to (\exists n.\ fn = \mathsf{true})\\
\mathsf{MP}_\mathsf{PR} &:= \forall f : \mathbb{N} \to \mathbb{B}.\ \mathsf{primitive\text{-}recursive}\ f \to && \neg\neg(\exists n.\ fn = \mathsf{true}) \to (\exists n.\ fn = \mathsf{true})
\end{aligned}$$

We write MP_PR following Troelstra and van Dalen [12]. Due to the Kleene normal form theorem [7], any principle replacing primitive recursiveness with computability in any Turing complete model is equivalent, e.g. called MP_L in [4] after the weak call-by-value λ-calculus L [6].

Note that $\mathsf{MP}_\mathbb{P}$ implies $\mathsf{MP}_\mathbb{B}$, which in turn implies MP_PR. The first implication is an equivalence given the axiom of (type-theoretic) unique choice, i.e. if $\forall R : \mathbb{N} \to \mathbb{B} \to \mathbb{P}.\ (\forall n : \mathbb{N}.\exists! b : \mathbb{B}. Rnb) \to \exists f : \mathbb{N} \to \mathbb{B}.\forall n.\ Rn(fn)$ holds, because then any such $A : \mathbb{N} \to \mathbb{P}$ gives rise to a decider of type $\mathbb{N} \to \mathbb{B}$. The second implication is an equivalence under CT ("Church's thesis" [9]), i.e. if the proposition $\forall f : \mathbb{N} \to \mathbb{B}.\ \mathsf{computable}\ f$ holds. $\mathsf{MP}_\mathbb{P}$ is consistent because it is a consequence of the law of excluded middle (LEM). $\mathsf{MP}_\mathbb{B}$ is proved independent from type theory by Mannaa and Coquand [2] as well as Pedrót and Tabareau [10], and MP_L by Forster, Kirst, and Wehr [5]. In a (weak) type theory such as CIC, both the unique choice axiom from above and CT are independent. In many constructive foundations (CZF, IZF, HoTT, or in MLTT with $\exists$ as Σ), unique choice is a theorem, but CT remains independent. Since in all these foundations $\exists n : \mathbb{N}.\ fn = \mathsf{true}$ implies $\Sigma n : \mathbb{N}.\ fn = \mathsf{true}$, stating MP with Σ or $\exists$ is equivalent.

On Σ^0_1 Predicates A predicate $p : X \to \mathbb{P}$ is stable under double negation if $\forall x.\ \neg\neg px \to px$, and is Σ^0_1 if there exists a decidable predicate $A : X \to \mathbb{N} \to \mathbb{P}$ such that $\forall x.\ px \leftrightarrow \exists n.\ Axn$. Now if decidable predicates $A : X \to \mathbb{N} \to \mathbb{P}$ only need to fulfill $\forall xn.\ Axn \lor \neg Axn$, then stability of Σ^0_1 predicates is equivalent to $\mathsf{MP}_\mathbb{P}$. If however decidable predicates are associated with a function of type $X \to \mathbb{N} \to \mathbb{B}$, stability of Σ^0_1 predicates is equivalent to $\mathsf{MP}_\mathbb{B}$. And if decidable predicates are associated with a *computable* function of that type, it is equivalent to MP_PR.

Post's Theorem (PT) [11] states that Σ_1^0 predicates with complement in Σ_1^0 are decidable. With decidable predicates defined using type-theoretic functions, PT is equivalent to $\mathsf{MP}_{\mathbb{B}}$ [12], formalised in Coq by Forster, Kirst, and Smolka [3]. With decidable predicates defined using computable functions, PT is equivalent to $\mathsf{MP}_{\mathsf{PR}}$, formalised in Coq by Forster and Smolka [6]. With the logical definition, PT is equivalent to $\mathsf{MP}_{\mathbb{P}}$, a proof we contribute with this abstract.

Termination of Computation It is folklore that "a computation halts if it does not run forever" is equivalent to MP. Taking a computation as a Σ_1^0 relation $\mathbb{N}\to\mathbb{N}\to\mathbb{P}$, the three respective definitions of Σ_1^0 indeed render this equivalent to the respective version of MP. In particular, the statement "a Turing machine halts if it does not run forever" is equivalent to $\mathsf{MP}_{\mathsf{PR}}$.

Extended Natural Numbers One can model the extension of $\mathbb{N}$ with a point of infinity as monotonous infinite sequences of truth values b_i (if b_i then b_j holds for $j \geq i$). MP is equivalent to "an extended natural number which is not infinite is finite", precisely to $\mathsf{MP}_{\mathbb{P}}$ if sequences are defined as predicates $\mathbb{N} \to \mathbb{P}$, and to $\mathsf{MP}_{\mathbb{B}}$ if defined as functions $\mathbb{N} \to \mathbb{B}$. Defining sequences as *computable* functions $\mathbb{N} \to \mathbb{B}$ is unusual, but would be equivalent to $\mathsf{MP}_{\mathsf{PR}}$.

First-order Completeness It was already known to Gödel that completeness of natural deduction w.r.t. Tarski-semantics over the $(\forall, \to, \bot)$-fragment of classical first-order logic is equivalent to $\mathsf{MP}_{\mathsf{PR}}$ [8]. The result can be extended to Σ_1^0-theories, but again the definition of Σ_1^0 is crucial. The equivalences to $\mathsf{MP}_{\mathbb{B}}$ and $\mathsf{MP}_{\mathsf{PR}}$ are proved in Coq by Forster, Kirst, and Wehr [4], we contribute the respective (Coq) proof for $\mathsf{MP}_{\mathbb{P}}$.

$\mathbf{TT}_{\mathcal{C}}^{\square}$ is a general framework for type theories modeled through an abstract modality $\square$ and parameterised by a type of time-progressing choice operators $\mathcal{C}$ due to Cohen and Rahli [1], which is formalised in Agda. Time-progression here means that $\mathrm{TT}_{\mathcal{C}}^{\square}$'s computation system includes stateful computations that can evolve non-deterministically over time (captured by a poset $\mathcal{W}$ of worlds), and that can change the state of the world. Instantiating $\square$ and $\mathcal{C}$ can either validate or invalidate axioms such as MP.

Separation of $\mathsf{MP}_{\mathbb{B}}$ and $\mathsf{MP}_{\mathsf{PR}}$ We prove that instantiating $\mathcal{C}$ with choice sequences and $\square$ with a Beth modality as in [1] yields a model validating constructively $\neg\mathsf{MP}_{\mathbb{B}}$, and, assuming LEM in the meta-theory, $\mathsf{MP}_{\mathsf{PR}}$. To do so, we translate the types $\mathbb{N}$ and $\mathbb{B}$ to the types Nat and Bool of possibly effectful terms with two properties: (1) if they compute to a value in a world, they compute to the same value in all extensions of that world; and (2) whenever they compute to a value, they leave the world unchanged. Such effectful terms do not satisfy $\mathsf{MP}_{\mathbb{B}}$ because f can be undetermined for all inputs and thus satisfy $\neg\neg(\exists n.\ fn = \mathsf{true})$ but not $\exists n.\ fn = \mathsf{true}$. However, primitive recursive functions can be encoded as natural numbers, and thus behave like a pure, effect-free function. Concretely, we have that

$$\forall (w : \mathcal{W}).w \not\models \mathbf{\Pi} f{:}\mathsf{Nat} \to \mathsf{Bool}.(\neg\neg{\downarrow}\mathbf{\Sigma} n{:}\mathsf{Nat}.f\ n = \mathsf{true}) \to {\downarrow}\mathbf{\Sigma} n{:}\mathsf{Nat}.f\ n = \mathsf{true}$$

Here, the $\downarrow$ operation discards the computational content of the dependent pair type $\mathbf{\Sigma}$. Furthermore, with LEM in the meta-theory, MP for *pure* (i.e. effect-free) functions is valid in *all* models in [1] (see mpp.lagda), with $\mathbf{\Pi_p}$ letting f range over pure, effect-free terms only:

$$\forall (w : \mathcal{W}).w \models \mathbf{\Pi_p} f{:}\mathsf{Nat} \to \mathsf{Bool}.(\neg\neg{\downarrow}\mathbf{\Sigma} n{:}\mathsf{Nat}.f\ n = \mathsf{true}) \to {\downarrow}\mathbf{\Sigma} n{:}\mathsf{Nat}.f\ n = \mathsf{true}$$

To show that this implies $\mathsf{MP}_{\mathsf{PR}}$, note that $\mathsf{MP}_{\mathsf{PR}}$ can be equivalently stated as

$$\forall (w : \mathcal{W}).w \models \mathbf{\Pi} m{:}\mathsf{Nat}.(\neg\neg{\downarrow}\mathbf{\Sigma} n{:}\mathsf{Nat}.\mathsf{eval}\ m\ n = \mathsf{true}) \to {\downarrow}\mathbf{\Sigma} n{:}\mathsf{Nat}.\mathsf{eval}\ m\ n = \mathsf{true}$$

where $\mathsf{eval} : \mathbb{N} \to \mathbb{N} \to \mathbb{B}$ is a pure function which interprets its first argument as the Gödelisation of a primitive recursive function f, and for any primitive recursive f there is m with $\forall n.fn = \mathsf{eval}\ m\ n$. Now whenever an effectful m evaluates to c in a world w, we have that $\mathsf{eval}\ c$ is a pure function for all $w' \sqsupseteq w$, making the pure form of MP applicable (see pure2.lagda).

Acknowledgements We have posed the question how to prove that $\mathsf{MP}_{\mathbb{B}}$ and $\mathsf{MP}_{\mathsf{PR}}$ are not equivalent to many people over the years, and want to thank Liron Cohen, Thierry Coquand, Hugo Herbelin, Hajime Ishihara, Pierre-Marie Pédrot, and Gert Smolka for discussions and useful hints about directions to chase. Furthermore, Martín Escardó provided useful feedback on a first draft of this abstract and the idea to extend the question to $\mathsf{MP}_{\mathbb{P}}$ came up in a discussion with Dirk Pattinson. Thank you!

References

[1] Liron Cohen and Vincent Rahli. Constructing unprejudiced extensional type theories with choices via modalities. In *FSCD 2022*, volume 228 of *LIPIcs*, pages 10:1–10:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[2] Thierry Coquand and Bassel Mannaa. The independence of markov's principle in type theory. *Log. Methods Comput. Sci.*, 13(3), 2017.

[3] 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 - CPP 2019*. ACM Press, 2019.

[4] Yannick Forster, Dominik Kirst, and Dominik Wehr. Completeness theorems for first-order logic analysed in constructive type theory. In *International Symposium on Logical Foundations of Computer Science*, pages 47–74. Springer, 2020.

[5] Yannick Forster, Dominik Kirst, and Dominik Wehr. Completeness theorems for first-order logic analysed in constructive type theory (extended version). *Journal of Logic and Computation*, 31(1):112–151, January 2021.

[6] 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.

[7] S. C. Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53(1):41–73, 1943.

[8] Georg Kreisel. On weak completeness of intuitionistic predicate logic. *J. Symb. Log.*, 27(2):139–158, 1962.

[9] Georg Kreisel. Mathematical logic. *Lectures in modern mathematics*, 3:95–195, 1965.

[10] Pierre-Marie Pédrot and Nicolas Tabareau. Failure is not an option. In *European Symposium on Programming*, pages 245–271. Springer, 2018.

[11] Emil L. Post. Recursively enumerable sets of positive integers and their decision problems. *bulletin of the American Mathematical Society*, 50(5):284–316, 1944.

[12] Anne Sjerp Troelstra and Dirk van Dalen. Constructivism in mathematics. vol. i. *Studies in Logic and the Foundations of Mathematics*, 26, 1988.

7

Session 9: Formalizing mathematics using type theory

Classification of Covering Spaces and Canonical Change of Basepoint

Jelle Wemmenhove[1], Cosmin Manea[2], and Jim Portegies[1]

[1] Eindhoven University of Technology, Eindhoven, The Netherlands
[2] ETH Zurich - Swiss Federal Institute of Technology, Zürich, Switzerland

Homotopy Type Theory is an extension of MLTT that allows us to study ideas from topology by re-expressing them a synthetic language, see [Uni13, LB13, HFFLL16]. For example, paths are modeled by terms of equality types. The synthetic nature allows one to manipulate the nuts and bolts of homotopy theory directly, keeping a strong connection to the geometric ideas.

With the desire to gain better insight into how to approach homotopy theory in HoTT, we set out to prove that there is no degree one map from a closed oriented genus g surface to a closed oriented genus h surface if $g < h$. Although we have not reached that destination, along the way we have proved synthetic versions of some classical results.

Building on Hou (Favonia) and Harper's results on covering spaces [HH18], we prove HoTT versions of the lifting criterion and the classification of covering spaces; although these are already shown in HoTT [BvDR18, Thrm. 7], the proofs we provide are more basic and might be more accessible. Secondly, we show when there exists canonical change-of-basepoint isomorphisms $\pi_n(X,a) \cong \pi_n(X,b)$. The theory is formalized in Coq using the HoTT library [BGL+17], see `https://gitlab.tue.nl/computer-verified-proofs/covering-spaces`.

Classification of Covering Spaces. Before one can prove results from classical homotopy theory in HoTT, one needs to find proper translations of these results. One might try a direct translation first: the classical results can be stated in HoTT using existing definitions of the fundamental group and induced maps from the HoTT book [Uni13]. The new language also allows one to express the underlying ideas in different ways.

Using Hou (Favonia) and Harper's definition of pointed covering spaces in HoTT [HH18, Def. 7], we prove a direct translation of the lifting criterion.

Lemma 1 (Lifting Criterion, cf. [Hat01, Prop. 1.33])**.** *Let $F : X \to \mathsf{Set}$ with $u_0 : F(x_0)$ be a pointed covering space over a pointed type (X, x_0) . A pointed map $f : (Y, y_0) \cdot\!\!\to (X, x_0)$, with Y a connected type, can be lifted to a map $\tilde{f} : (Y, y_0) \cdot\!\!\to (\sum_X F, (x_0, u_0))$ if and only if*

$$f_*(\pi_1(Y, y_0)) \subset \mathsf{pr_1}_*(\pi_1(\textstyle\sum_X F, (x_0, u_0))) \ . \tag{1}$$

Here f_* and $\mathsf{pr_1}_*$ denote the induced maps on the fundamental groups, not the shorthand notation for transport as is common in HoTT.

In proving this statement, we found criterion (1) inconvenient to work with in HoTT. The notation $f_*(\pi_1(Y, y_0))$, for example, conceals multiple truncations — a propositional-truncation to define the image of a map and a set-truncation for π_1 — which hinder access to the homotopical objects. We therefore proved that criterion (1) is equivalent to another condition, one tailored to HoTT.

Lemma 2. *Let $F : X \to \mathsf{Set}$ with $u_0 : F(x_0)$ be a pointed covering space over a pointed type (X, x_0) and let $f : (Y, y_0) \cdot\!\!\to (X, x_0)$ be a pointed map. Then the criterion $f_*(\pi_1(Y, y_0)) \subset \mathsf{pr_1}_*(\sum_X F, (x_0, u_0))$ is equivalent to the condition that for all loops $p : y_0 =_Y y_0$ there exists a loop from u_0 to u_0 lying over $f_*(p)$ in F , meaning that*

$$\mathsf{transport}^F(f_*(p), u_0) =_{F(x_0)} u_0 \ .$$

Together with the universal covering space constructed by Hou (Favonia) and Harper [HH18, Thrm. 13], we use the alternative lifting criterion (Lemma 2) to show that connected, pointed covering spaces are classified by subgroups of $\pi_1(X, x_0)$.

Theorem 3 (Classification, cf. [Hat01, first half of Thm. 1.38])**.** *Let (X, x_0) be a connected, pointed type. Then there is an equivalence between pointed, connected covering spaces (F, u_0) over (X, x_0) and subgroups of $\pi_1(X, x_0)$, obtained by associating to the covering space (F, u_0) the subgroup given by the predicate*

$$|p|_0 \mapsto \left(\mathsf{transport}^F(p, u_0) =_{F(x_0)} u_0\right) ,$$

meaning that $|p|_0 : \pi_1(X, x_0)$ belongs to the subgroup if there exists a loop from u_0 to u_0 lying over p.

Canonical Change of Basepoint. In classical homotopy theory, a path p from a to b in a topological space X induces a change-of-basepoint isomorphism between homotopy groups $\pi_n(X, a) \cong \pi_n(X, b)$. The isomorphism depends on the homotopy class of the path p. In the case that X is simply-connected, the isomorphism can be considered canonical — there is only one homotopy class of paths from a to b.

In HoTT, transport along a path $p : a =_X b$ also gives rise to an isomorphism $\pi_n(X, a) \cong \pi_n(X, b)$. Often we do not have access to an explicit path $p : a =_X b$, but only know the truncation $\|a =_X b\|$ to be inhabited. In these cases, we can use *extension by weak constancy* [HH18, Lemma 6]: there exists a canonical isomorphism $\pi_n(X, a) \cong \pi_n(X, b)$ if transport along all paths $p, q : a =_X b$ yields the same results, i.e.

$$\mathsf{transport}^{\pi_n(X,-)}(p, -) \;=\; \mathsf{transport}^{\pi_n(X,-)}(q, -) \;.$$

This is equivalent to stating that the fundamental group $\pi_1(X, a)$ acts trivially on the higher homotopy groups $\pi_n(X, a)$.

We prove the following statements for when the π_1-action is trivial, and hence for when there exists canonical change-of-basepoint isomorphisms.

Theorem 4. *Let X be a type with designated point $a : X$.*

(i) If X is simply-connected, then the action of $\pi_1(X, a)$ on $\pi_n(X, a)$ is trivial for all $n \geq 1$;

(ii) The fundamental group $\pi_1(X, a)$ is abelian if and only if the action on itself is trivial;

(iii) If the type $\prod_{p,q\,:\,\Omega(X,a)} p \cdot q = q \cdot p$ is merely inhabited, then the action of $\pi_1(X, a)$ on $\pi_n(X, a)$ is trivial for all $n \geq 1$.

Results (i) and (ii) are easily shown, both in the classical theory and in HoTT. For result (iii), we use the relationship between transport in consecutive loop spaces given below; it follows from [Uni13, Thrm. 2.11.4]. Loop spaces that satisfy the assumption in (iii) are classically called *homotopy* commutative as the commutativity may only hold up to homotopy; this is the default setting in HoTT.

Lemma 5. *Let $p : a =_X b$ and $u : \Omega^{n+1}(X, a)$, then in $\Omega^{n+1}(X, b)$ we have that:*

$$\mathsf{transport}^{\Omega^{n+1}(X,-)}(p, u) = (\mathsf{apd}_{\mathsf{refl}^n_{(-)}}(p))^{-1} \cdot \mathsf{ap}_{\mathsf{transport}^{\Omega^n(X,-)}(p,-)}(u) \cdot \mathsf{apd}_{\mathsf{refl}^n_{(-)}}(p) \;.$$

The term $\mathsf{apd}_{\mathsf{refl}^n_{(-)}}(p)$ can be thought of as a homotopy between the transported n-cell $p_*(\mathsf{refl}^n_a)$ at b and the constant n-cell refl^n_b at b.

References

[BGL$^+$17] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The hott library: A formalization of homotopy type theory in coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, page 164–172, New York, NY, USA, 2017. Association for Computing Machinery.

[BvDR18] Ulrik Buchholtz, Floris van Doorn, and Egbert Rijke. Higher groups in homotopy type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 205–214, New York, NY, USA, 2018. Association for Computing Machinery.

[Hat01] Allen Hatcher. *Algebraic Topology.* Cambridge University Press, 2001.

[HFFLL16] 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 *2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–10, 2016.

[HH18] Kuen-Bang Hou (Favonia) and Robert Harper. Covering Spaces in Homotopy Type Theory. In Silvia Ghilezan, Herman Geuvers, and Jelena Ivetić, editors, *22nd International Conference on Types for Proofs and Programs (TYPES 2016)*, volume 97 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[LB13] Daniel R. Licata and Guillaume Brunerie. $\pi_n(S^n)$ in homotopy type theory. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, pages 1–16, Cham, 2013. Springer International Publishing.

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

Choreographic Programming in Coq

Luís Cruz-Filipe, Lovro Lugović, Fabrizio Montesi, Marco Peressotti, and Robert R. Rasmussen *

Department of Mathematics and Computer Science, University of Southern Denmark
{lcf,lugovic,fmontesi,peressotti,rrr}@imada.sdu.dk

Choreographic programming is a paradigm for specifying concurrent systems (choreographies) based on message-passing where communications are written in an Alice-to-Bob notation. Choreographies can be mechanically *projected* into distributed process-calculus implementations guaranteed to be bisimilar to the original choreography. Such implementations can never suffer from mismatched communications – more generally, they cannot reach a deadlocked state, as the syntax of choreography language cannot express deadlocks.

Example 1 (From [8])**.** *The following choreography models a scenario where Alice (*a*) buys a book from a seller (*s*) routing the payment through her bank (*b*).*

$$\begin{aligned}&\mathsf{a}.\mathit{title} \to \mathsf{s};\ \mathsf{s}.\mathit{price} \to \mathsf{a};\ \mathsf{s}.\mathit{price} \to \mathsf{b};\\ &\mathit{if}\ \mathsf{b}.\mathit{approves}\ \mathit{then}\ \mathsf{b} \to \mathsf{s}[\mathit{ok}];\ \mathsf{b} \to \mathsf{a}[\mathit{ok}];\ \mathsf{s}.\mathit{book} \to \mathsf{a}\\ &\qquad\quad \mathit{else}\ \mathsf{b} \to \mathsf{s}[\mathit{ko}];\ \mathsf{b} \to \mathsf{a}[\mathit{ko}];\ \mathbf{0}\end{aligned}$$

First, Alice sends the title of the book to the seller, who quotes the price to both Alice and the bank. The bank can then confirm the transaction by sending an acknowledgement to both Alice and the seller (after which the latter sends the book), or send a cancellation to both parties.

This choreography can be projected into the following distributed protocol.

$$\begin{aligned}\mathsf{a}\ &\triangleright\ \mathsf{s}!\mathit{title};\ \mathsf{s}?;\ \mathsf{b}\&\{\mathit{ok}:\ \mathsf{s}?, \mathit{ko}:\ \mathbf{0}\}\\ \mathsf{b}\ &\triangleright\ \mathsf{s}?;\ \mathit{if}\ \mathit{approves}\ \mathit{then}\ (\mathsf{s} \oplus \mathit{ok};\ \mathsf{a} \oplus \mathit{ok})\ \mathit{else}\ (\mathsf{s} \oplus \mathit{ko};\ \mathsf{a} \oplus \mathit{ko})\\ \mathsf{s}\ &\triangleright\ \mathsf{a}?;\ \mathsf{a}!\mathit{price};\ \mathsf{b}!\mathit{price};\ \mathsf{b}\&\{\mathit{ok}:\ \mathsf{a}!\mathit{book}, \mathit{ko}: \mathbf{0}\}\end{aligned}$$

Alice's protocol is: send a title to the seller and wait for a reply; then wait for either confirmation from the bank, in which case the seller sends the book, or cancellation, in which case the protocol ends. The protocol for the seller is similar. In turn, the bank initially waits for a message from the seller, and then decides whether to send confirmation or cancellation to the seller and Alice.

The precise operational correspondence between choreographies and their projections (the *EPP Theorem*) ensures that the choreography and the distributed protocol in the example above behave in the same way. The proof of this result is complex, due to the high number of cases that need to be considered and to the multitude of rules in the semantics of both choreography and process languages. Such proofs are prone to errors when designed and checked by humans: a previous attempt to formalise a higher-order process calculus [15] turned up a number of problems in the original proofs [16]; similar issues have arisen in the field of multiparty session types [20, Section 8.1], closely related to choreographic programming.

These issues motivated a subset of the present authors to formalise the theory of choreographic programming in the theorem prover Coq [8]. The result was a formalisation of a core model for choreographic programming [6, 17], including the choreographic language and proof

*This work was partially supported by Villum Fonden, grants 29518 and 50079, and Independent Research Fund Denmark, grant 0135-00219.

of its Turing completeness [10] and the target language for distributed implementations [6] together with a proof of the EPP Theorem [9]. In those works, we stated that our formalisation was developed with an intent to be extendable and flexible. In this abstract, we report on recent developments that build upon this formalisation, using it as an effective research tool, thereby establishing its reusability and its usefulness.

Choreography amendment. Not all choreographies can be projected to distributed implementations, because of a realisability requirement known in the field as *knowledge of choice* [3]. Essentially, this condition means that every process whose behaviour depends on a conditional expression evaluated by another process must be notified of this result. In the example above, this is achieved by the *label selection* communications, e.g., $\mathsf{b} \to \mathsf{s}[ok]$.

Amendment is a transformation that makes every choreography projectable by adding such label selections where needed. This procedure is described in [6]; however, the main correspondence result between the semantics of the original and the amended choreography is wrong. The error was only discovered while formalising the proof, and the counterexamples found with the theorem prover's help were essential to establishing and proving the correct correspondence [7].

Livelocks. Requiring knowledge of choice disallows choreographies where some participants are inactive while others engage in a loop – for example, if Alice and the seller engage in a negotiation until they agree on a price, after which the bank is notified of the amount to transfer. A direct formalisation of this protocol as a choreography would not be projectable: knowledge of choice would require the bank to be informed of the result of each iteration. This constraint is unreasonable in practice.

In recent work [11], we have relaxed this requirement to allow for projecting several new scenarios that occur in practice. The use of the theorem prover was again essential to detect edge cases that were not found while making pen-and-paper proofs. This development also supports the claim of modularity of the formalisation, as the proof of the EPP Theorem was mostly unchanged as soon as the relevant lemmas had been generalised to the new notion of projection.

Compilation. The distributed implementations generated by choreographic programming are written in a mathematical process language. However, they are close enough to implementation languages that they can be very directly translated to executable code.

We have implemented a toolchain that allows users to write choreographies, translates them in Coq terms, applies the projection procedure extracted from the Coq formalisation to obtain a distributed process implementation, and finally compiles this implementation into executable code [5]. The final compilation step is done by a handwritten program, but since it is completely homeomorphic (in the sense that each process action is modularly translated to Jolie code [18]) its correctness is easy to establish without requiring a full formalisation.

Related work. After our initial work, other groups have developed formalisations of choreographic and related languages. Kalas [19] is a certified compiler written in HOL from a choreographic language similar to ours to CakeML, with an asynchronous semantics but a more restricted notion of projection. In particular, processes evaluating conditionals must immediately send selections to the processes that need them, while our language is more faithful to the pen-and-paper literature on choreographies [1, 2, 13].

Pirouette [12] is a functional choreographic programming language formalised in Coq, supporting asynchronous communication and higher-order functions. These capabilities come at the cost of hidden global synchronisations, while our language is fully decentralised, with all synchronisations syntactically explicit.

Another related line of research is that of multiparty session types [13], which can be seen as choreographies without computation – and therefore simpler. There are two available formalisations of multiparty session types [4, 14], which include a counterpart to the EPP theorem, but are even more restrictive than Kalas in how they project conditionals.

References

[1] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012.

[2] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *Procs. POPL*, pages 263–274. ACM, 2013.

[3] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Log. Methods Comput. Sci.*, 8(1), 2012.

[4] David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. Zooid: a DSL for certified multiparty computation: from mechanised metatheory to certified multiparty processes. In Stephen N. Freund and Eran Yahav, editors, *Procs. PLDI*, pages 237–251. ACM, 2021.

[5] Luís Cruz-Filipe, Lovro Lugović, and Fabrizio Montesi. Certified compilation of choreographies with hacc. *CoRR*, abs/2303.03972, 2023. Accepted for publication at *FORTE'23*.

[6] Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020.

[7] Luís Cruz-Filipe and Fabrizio Montesi. Now it compiles! certified automatic repair of uncompilable protocols. *CoRR*, abs/2302.14622, 2023. Accepted for publication at *ITP'23*.

[8] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Choreographies in Coq. In *TYPES 2019, Abstracts*, 2019. Extended abstract.

[9] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Procs. ICTAC*, volume 12819 of *LNCS*, pages 115–133. Springer, 2021.

[10] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Formalising a Turing-complete choreographic language in Coq. In Liron Cohen and Cezary Kaliszyk, editors, *Procs. ITP*, volume 193 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[11] Luís Cruz-Filipe, Fabrizio Montesi, and Robert R. Rasmussen. Keep me out of the loop: a more flexible choreographic projection. Accepted for publication., 2023.

[12] Andrew K. Hirsch and Deepak Garg. Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022.

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

[14] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Multiparty gv: Functional multiparty session types with certified deadlock freedom. In *Procs. ICFP*, 2022. Accepted for publication.

[15] Ivan Lanese, Jorge A. Pérez, Davide Sangiorgi, and Alan Schmitt. On the expressiveness and decidability of higher-order process calculi. *Inf. Comput.*, 209(2):198–226, 2011.

[16] Petar Maksimovic and Alan Schmitt. HOCore in Coq. In Christian Urban and Xingyuan Zhang, editors, *ITP*, volume 9236 of *LNCS*, pages 278–293. Springer, 2015.

[17] Fabrizio Montesi. *Introduction to Choreographies.* Cambridge University Press, 2023.

[18] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Web Services Foun-*

dations, pages 81–107. Springer, 2014.

[19] Johannes Åman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. Kalas: A verified, end-to-end compiler for a choreographic language. In June Andronick and Leonardo de Moura, editors, *Procs. ITP*, volume 237 of *LIPIcs*, pages 27:1–27:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[20] Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *PACMPL*, 3(POPL):30:1–30:29, 2019.

Dynamic Type Theory

Matthias Eberl

LMU Munich, Germany
`matthias.eberl@mail.de`

Abstract

Dynamic type theory is a model of type theory which is based on the potentialist's point of view. It has stages for each type and a refinement of type declarations, called state declarations, defining input and output bounds.

1 Introduction

The starting point is to understand mathematics from the potential infinite without the concept of an actual, i.e., completed infinity. The potential infinite is thereby seen as a dynamic concept, as an extensible finite. Seen in this way, the approach leads to finitism. However, the concepts are not tied to this finitistic view — no notion of finiteness is required for their development[1]. The main idea is that extensibility is more fundamental than completeness. This unconditional extensibility includes Dummett's notion of indefinite extensibility, i.e., the phenomenon that a reference to all objects of a concept leads to a new object of that concept. This is possible because the interpretation of the universal quantifier uses a reflection principle.

A dynamic type theory naturally has a hierarchy of (Grothendieck) universes, since extensibility and stages are built in from the very beginning. The approach is model theoretic, so there are (almost — see Section 3 below) no restrictions in proof theory. This means that classical logic is applicable as well as intuitionistic logic.

2 Sets and Systems

The basis for a dynamic model of type theory are stages and we assume that each type ϱ has its own set $\mathcal{I}_\varrho$ of stages. The sets $\mathcal{I}_\varrho$ are endowed with an order $\leq$ that makes them directed sets. From a consequent type theoretic perspective however, $\mathcal{I}_\varrho$ will be a type ϱ^* and indices i will be terms of type ϱ^*. We complement the static interpretation of type ϱ with a system of finite sets. Such a system is a family[2] $(\mathcal{M}_i)_{i:\varrho^*}$ with finite sets $\mathcal{M}_i$ and a relation $\overset{p}{\mapsto}$ between states $a_{i'} \in \mathcal{M}_{i'}$ and $a_i \in \mathcal{M}_i$, indicating that a_i is a predecessor of $a_{i'}$ whenever $a_{i'} \overset{p}{\mapsto} a_i$.[3] The main point of a dynamic interpretation is that any reference to the interpretation of ϱ can only be made by first referring to a stage $i : \varrho^*$ and then using states a_i of objects in $\mathcal{M}_i$.

For instance, the index set of type *nat* (natural numbers), is $\mathbb{N}^+$ (set of positive natural numbers). The interpretation of the universal quantifier over *nat* has to choose an index $i \in \mathbb{N}^+$ and use the set $\mathbb{N}_i := \{0, \ldots, i-1\}$ as its domain. The idea is to choose a *sufficiently large* index as presented in [1] for first-order logic (based on [5] and [4]). In short, if the formula has free variables $\mathsf{x}_0, \ldots, \mathsf{x}_{n-1}$, then the assignment $\boldsymbol{a}$ is taken from $\mathcal{M}_{i_0} \times \cdots \times \mathcal{M}_{i_{n-1}}$ and the index satisfies $i \gg C := (i_0, \ldots, i_{n-1})$, i.e., i is sufficiently large relative to C.

[1]The finitistic view, however, is philosophically most convincing (see e.g. [3]), in particular it avoids all paradoxes of the infinite.

[2]We prefer $i : \varrho^*$ instead of $i \in \mathcal{I}_\varrho$ because there is no assumption that the index set exists as a complete set.

[3]For details see [2], also available at my homepage https://www.indefinitely-extensible.com.

An important concept for a higher-order logic, and for type theory, is the notion of a limit of a system, see [2]. The relation between the dynamic system and the static limit has two readings, depending on one's meta-level assumptions. If one assumes that actual infinite sets exist, then the limit of a system is such an infinite set, e.g., the limit of $(\mathbb{N}_i)_{i:nat^+}$ is the infinite set $\mathbb{N}$. From a consequent finitistic point of view, a limit is a sufficiently large state $\mathbb{N}_j$ of the system itself that reflects all properties of such an ideal (but non-existent) infinite limit set $\mathbb{N}$. The concrete instance of such a finite set depends on (the stage of) the investigation.

3 Type Theoretic Concepts

We use types over arbitrary base types ι, including types *nat* and *bool* (Boolean values *true* and *false*), and a function type constructor, so that types are $\varrho ::= \iota \mid \varrho \to \varrho$. An interpretation of a type and a term has two parts, a dynamic part and a static part, whereby the dynamic part is a system and the static part is the limit of this system. To define the dynamic part of the interpretation of λ-terms we need to restrict the type declarations to contexts $\Gamma = (\varrho_0, \ldots, \varrho_{n-1})$ where variables are either positive or negative.[4] They are defined by $Typ^+ \ni \varrho^+ ::= \iota \mid (\varrho^- \to \varrho^+)$ and $Typ^- \ni \varrho^- ::= bool \mid (\varrho^+ \to \varrho^-)$. Positive types correspond to objects, negative types to properties on these objects. Terms then have a type in $Typ^1 \ni \varrho^1 ::= \iota \mid (\varrho^+ \to \varrho^1) \mid (\varrho^- \to \varrho^1)$, which are functions on objects and properties. Parallel to the usual *type declaration* there is a refinement, called *state declaration*, defining the input and output bounds. Let i, j be stages and $C = (i_0, \ldots, i_{n-1})$:

$$(\textsc{Var})\ \frac{}{\Gamma \vdash \mathsf{x_k} : \varrho_k} \qquad (\textsc{Var}+)\ \frac{j \geq i_k \quad \varrho_k \in Typ^+}{C \vdash \mathsf{x_k} : j} \qquad (\textsc{Var}-)\ \frac{j \leq i_k \quad \varrho_k \in Typ^-}{C \vdash \mathsf{x_k} : j}$$

$$(\textsc{App})\ \frac{\Gamma \vdash \mathsf{r} : \varrho \to \sigma \qquad \Gamma \vdash \mathsf{s} : \varrho}{\Gamma \vdash \mathsf{rs} : \sigma} \qquad (\textsc{App})\ \frac{C \vdash \mathsf{r} : i \to j \qquad C \vdash \mathsf{s} : i}{C \vdash \mathsf{rs} : j}$$

$$(\textsc{Lam})\ \frac{\Gamma\varrho \vdash \mathsf{r} : \sigma}{\Gamma \vdash \lambda \mathsf{x}^{\varrho}_{\mathsf{n}}\mathsf{r} : \varrho \to \sigma} \qquad (\textsc{Lam})\ \frac{Ci \vdash \mathsf{r} : j}{C \vdash \lambda \mathsf{x}^{\varrho}_{\mathsf{n}}\mathsf{r} : i \to j}$$

This fragment of simple type theory allows term constructors $\mathsf{c} : (\sigma_0, \ldots, \sigma_{m-1}) \to \sigma$ for types $\sigma_0, \ldots, \sigma_{m-1}, \sigma \in Typ^1$, including universal quantifier $\forall_\varrho : (\varrho \to bool) \to bool$. Let $\Sigma^{\mathcal{I}}$ be a refinement of a signature Σ for states, then the rules are (here formulated only for the state declaration):

$$(\textsc{Cst})\ \frac{\mathsf{c} : (i_0, \ldots, i_{m-1}) \to i \in \Sigma^{\mathcal{I}} \qquad C \vdash \mathsf{r_0} : i_0 \ \ldots \ C \vdash \mathsf{r_{m-1}} : i_{m-1} \qquad [\text{ Conditions }]}{C \vdash \mathsf{c}\mathsf{r_0} \ldots \mathsf{r_{m-1}} : i}$$

The central theorem is a reflection principle stating that all objects and operations (i.e., all terms) have an interpretation in the limit as well as in some state of the system, reflecting the limit element. From the perspective that the limit is an infinite set, this means that every object and operation in the (infinite) limit has a counterpart in some (finite) state of the system. From a consequent finitistic perspective this infinite limit set is an indefinitely large (or sufficiently large) finite set.

[4] The reason is that totality of higher-order functionals is challenging in this approach.

References

[1] Matthias Eberl. A model theory for the potential infinite. *Reports on Mathematical Logic*, 57:3–30, 2022.

[2] Matthias Eberl. Higher-order concepts for the potential infinite. *Theoretical Computer Science*, 945:113667, 2023.

[3] Peter Fletcher. *Truth, proof and infinity.* Kluver Academic Publishers, Netherlands, 1989.

[4] Shaughan Lavine. *Understanding the infinite.* Harvard University Press, 2009.

[5] Jan Mycielski. Locally finite theories. *Journal of Symbolic Logic*, 51(1):59–62, 1986.

8

Session 10: Foundations of type theory and constructive mathematics

Towards an Interpretation of Inaccessible Sets 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 (MLTT) was extended by Setzer [10, 11] with a large universe type called a *Mahlo universe*, in order to provide MLTT with an analogue of some large cardinal property. A Mahlo universe V has a reflection property similar to the one of weakly Mahlo cardinals: V is closed under any function f on families of small types in V, i.e., any function f of type $\Sigma_{(x:\mathrm{V})}(\mathrm{T_V}x \to \mathrm{V}) \to \Sigma_{(x:\mathrm{V})}(\mathrm{T_V}x \to \mathrm{V})$ with the decoding function $\mathrm{T_V} : \mathrm{V} \to \mathrm{Set}$ for V.[1] The resulting system **MLM** is thus an instance of constructive systems extended with an analogue of some large set or some large set itself. Setzer's purpose of introducing **MLM** is to obtain an extension of MLTT whose proof-theoretic ordinal is slightly greater than the one of Kripke-Platek set theory with one recursively Mahlo ordinal, where proof-theoretic ordinals of systems enable to measure the strength of a system.

Another instance of constructive systems extended with some large set was formulated in the context of Aczel's constructive set theory **CZF** [1, 2, 3]. Rathjen, Griffor and Palmgren [9] introduced the system $\mathbf{CZF}_\pi$, which is an extension of **CZF** with the existence of Mahlo's inaccessible sets of all transfinite orders [7]. The main purpose of introducing $\mathbf{CZF}_\pi$ is a proof-theoretic one as well: Rathjen, Griffor and Palmgren also introduced an extension of MLTT called **MLQ**, and determined its proof-theoretic ordinal by verifying that $\mathbf{CZF}_\pi$ is interpretable in **MLQ**. Though Rathjen [8] formulated another extension of **CZF** with the existence of a Mahlo set and showed that this extension is interpretable in **MLM**, it is not known whether $\mathbf{CZF}_\pi$ is interpretable in **MLM**. Specifically, it is an open question how to interpret the transfinite hierarchy of inaccessible sets in $\mathbf{CZF}_\pi$ by using the reflection property of **MLM**.

Aim and Approach. We, as a step towards an interpretation of $\mathbf{CZF}_\pi$ in **MLM**, show that the hierarchy of the types $\mathcal{V}^\alpha_{(a,f)}$ in Rathjen-Griffor-Palmgren's interpretation of $\mathbf{CZF}_\pi$ can be defined by means of the Mahlo universe V in **MLM**. This hierarchy was defined as the type-theoretic counterpart of α-inaccessible sets in [9], and its construction was provided by the two types M and Q in **MLQ**. Roughly speaking, Q is an inductive type of codes for operators which gives universes closed under universe operators constructed previously, while M is a universe closed under operators in Q.

Our idea for defining $\mathcal{V}^\alpha_{(a,f)}$ in **MLM** is to replace M with the Mahlo universe V, and then formulate a higher-order universe operator $\mathrm{u}^{\mathbb{M}}$ which is able to take a family (b, g) of universe operators constructed previously as an argument. The type of $\mathrm{u}^{\mathbb{M}}(b, g)$ is $\Sigma_{(x:\mathrm{V})}(\mathrm{T_V}x \to \mathrm{V}) \to \Sigma_{(x:\mathrm{V})}(\mathrm{T_V}x \to \mathrm{V})$, and we use the reflection property of V with respect to $\mathrm{u}^{\mathbb{M}}(b, g)$ to construct the hierarchy of the types $\mathcal{V}^\alpha_{(a,f)}$.

The reflection property of V can be informally explained as follows. Put $\mathrm{Fam(V)} := \Sigma_{(x:\mathrm{V})}(\mathrm{T_V}x \to \mathrm{V})$. The Mahlo universe type $\mathrm{V} : \mathrm{Set}$ reflects any function on families of small types in V: for any $f : \mathrm{Fam(V)} \to \mathrm{Fam(V)}$, there are a subuniverse U_f of V and its code $\widehat{\mathrm{U}}_f$ in V with the decoding function $\widehat{\mathrm{T}}_f : \mathrm{T_V}\,\widehat{\mathrm{U}}_f \to \mathrm{V}$ such that U_f is closed under f and $\mathrm{T_V}\,\widehat{\mathrm{U}}_f = \mathrm{U}_f$

[1]Below we use the logical framework adopted in the proof assistant Agda.

holds. The universe operator u above a family of small types in V is a typical example of such an $f : \mathrm{Fam}(\mathrm{V}) \to \mathrm{Fam}(\mathrm{V})$: for any $(b, g) : \mathrm{Fam}(\mathrm{V})$, $u\ (b, g) = (\widehat{\mathrm{U}}_{(b,g)}, \widehat{\mathrm{T}}_{(b,g)})$ is the pair of a universe $\widehat{\mathrm{U}}_{(b,g)} : \mathrm{V}$ and its decoding function $\widehat{\mathrm{T}}_{(b,g)} : \mathrm{U}_{(b,g)} \to \mathrm{V}$ such that $\mathrm{U}_{(b,g)}$ has codes of b and $g\ c$ for any $c : \mathrm{T}_{\mathrm{V}} b$. Then, the reflection property of V gives a subuniverse of V being closed under the universe construction by u. Of course, this does not exhaust the largeness of a Mahlo universe, since V has such a subuniverse for *any* function on families of small types.

We define the higher-order universe operator $\mathrm{u}^{\mathbb{M}}$ mentioned above as follows. We first stipulate the type O of *first-order operators* and the type Fam(O) of *families of first-order operators* as $\mathrm{O} := \mathrm{Fam}(\mathrm{V}) \to \mathrm{Fam}(\mathrm{V})$ and $\mathrm{Fam}(\mathrm{O}) := \Sigma_{(x:\mathrm{V})} \mathrm{T}_{\mathrm{V}} x \to \mathrm{O}$, respectively. Let $(z, v) : \mathrm{Fam}(\mathrm{O})$ and $(x, y) : \mathrm{Fam}(\mathrm{V})$ be given. Next, a function

$$h : \Pi_{(w:\mathrm{Fam}(\mathrm{V}))} \Big(((\mathrm{N}_1 + \mathrm{T}_{\mathrm{V}} x) + \mathrm{T}_{\mathrm{V}} z) + \Sigma_{(w' : \mathrm{T}_{\mathrm{V}} z)} \mathrm{T}_{\mathrm{V}} \mathrm{p}_1(v\ w'\ w) \to \mathrm{V} \Big)$$

is defined by

$$h\, w\ (\mathrm{i}(\mathrm{i}(\mathrm{i}\, x_1))) = x \text{ with } x_1 : \mathrm{N}_1, \tag{1}$$
$$h\, w\ (\mathrm{i}(\mathrm{i}(\mathrm{j}\, x_2))) = y\, x_2 \text{ with } x_2 : \mathrm{T}_{\mathrm{V}} x, \tag{2}$$
$$h\, w\ (\mathrm{i}(\mathrm{j}\, y_1)) = \mathrm{p}_1(v\, y_1\, w) \text{ with } y_1 : \mathrm{T}_{\mathrm{V}} z, \tag{3}$$
$$h\, w\ (\mathrm{j}\ (y_1, z_1)) = \mathrm{p}_2(v\, y_1\, w)\, z_1 \text{ with } y_1 : \mathrm{T}_{\mathrm{V}} z \text{ and } z_1 : \mathrm{T}_{\mathrm{V}} \mathrm{p}_1(v\, y_1\, w), \tag{4}$$

where p_1 (resp. p_2) is the left projection (resp. the right projection) for pair types, and i (resp. j) is the left injection (resp. the right injection) for sum types. Put $f^{\mathbb{M}}[z, v, x, y] : \mathrm{O}$ as

$$f^{\mathbb{M}}[z, v, x, y] := \lambda w.(((\widehat{\mathrm{N}_1}_{\mathrm{V}}\ \widehat{+}_{\mathrm{V}}\ x)\ \widehat{+}_{\mathrm{V}}\ z)\ \widehat{+}_{\mathrm{V}}\ \widehat{\Sigma}_{\mathrm{V}}(z, (w')\mathrm{p}_1(v\ w'\ w)),\ h\, w).$$

We then define $\mathrm{u}^{\mathbb{M}} : \mathrm{Fam}(\mathrm{O}) \to \mathrm{O}$ as $\mathrm{u}^{\mathbb{M}}\ (z, v)\ (x, y) := (\widehat{\mathrm{U}}_{f^{\mathbb{M}}[z,v,x,y]}, \widehat{\mathrm{T}}_{f^{\mathbb{M}}[z,v,x,y]})$ by reflecting $f^{\mathbb{M}}[z, v, x, y]$ in V. This reflection gives a subuniverse $\widehat{\mathrm{U}}_{f^{\mathbb{M}}[z,v,x,y]}$ closed under $f^{\mathbb{M}}[z, v, x, y]$: in particular, $\widehat{\mathrm{U}}_{f^{\mathbb{M}}[z,v,x,y]}$ is closed under each of first-order operators in $(z, v) : \mathrm{Fam}(\mathrm{O})$. Typical examples of such operators are universe operators constructed previously. Note that, for any $w : \mathrm{Fam}(\mathrm{V})$, we can extract from $f^{\mathbb{M}}[z, v, x, y]\ w$ a family of small types which is the result of applying an operator in (z, v) to w, as shown by (3) and (4) above.

The hierarchy of the types $\mathcal{V}^{\alpha}_{(a,f)}$ is constructed by iterating the operator $\mathrm{u}^{\mathbb{M}}$ along Aczel's iterative sets. In **MLM**, the type **V** of iterative sets is defined as $\mathbf{V} := \mathrm{W}_{(x:\mathrm{V})} \mathrm{T}_{\mathrm{V}} x$ with a standard definition of $\mathrm{index} : \mathbf{V} \to \mathrm{V}$ and $\mathrm{pred} : \Pi_{(x:\mathbf{V})} \mathrm{T}_{\mathrm{V}}(\mathrm{index} x) \to \mathbf{V}$ such that $\mathrm{index}\,(\sup a\, f) = a$ and $\mathrm{pred}\,(\sup a\, f) = f$ hold. We then prove the transfinite induction on the *transitive closure* α_{tc} of $\alpha : \mathbf{V}$, which is presupposed in [9]: $\mathrm{tcTI} : \Pi_{(\alpha:\mathbf{V})}(\Pi_{(x:\mathrm{T}_{\mathrm{V}}(\mathrm{index}\ \alpha_{\mathrm{tc}}))} F(\mathrm{pred}\, \alpha_{\mathrm{tc}} x) \to F\alpha) \to \Pi_{(\alpha:\mathbf{V})} F\alpha$. We define the function $\Phi : \mathbf{V} \to \mathrm{O}$ as $\Phi\alpha = \mathrm{tcTI}\ (\lambda\beta.\lambda x.\mathrm{u}^{\mathbb{M}}\ (\mathrm{index}\ \beta_{\mathrm{tc}}, x))\ \alpha$ by this induction principle. Roughly speaking, $\Phi\,\alpha$ means the iteration of $\mathrm{u}^{\mathbb{M}}$ along α. Finally, for any $a : \mathrm{V}$ and $f : \mathrm{T}_{\mathrm{V}} a \to \mathrm{V}$, we define the α*-th subuniverse* $\mathcal{M}^{\alpha}_{(a,f)}$ of V and the type $\mathcal{V}^{\alpha}_{(a,f)}$ of α*-th iterative sets* on $\mathcal{M}^{\alpha}_{(a,f)}$ as follows: $\mathcal{M}^{\alpha}_{(a,f)} := \mathrm{T}_{\mathrm{V}}(\mathrm{p}_1(\Phi\alpha(a, f)))$, $\mathrm{T}^{\alpha}_{(a,f)} := \lambda x.\mathrm{T}_{\mathrm{V}}(\mathrm{p}_2(\Phi\alpha(a, f))x)$, $\mathcal{V}^{\alpha}_{(a,f)} := \mathrm{W}_{(x:\mathcal{M}^{\alpha}_{(a,f)})} \mathrm{T}^{\alpha}_{(a,f)} x$. We formalised these definitions in Agda, using the external Mahlo universe introduced by [5].[2]

The next research direction is to formulate an interpretation of $\mathbf{CZF}_{\pi}$ in **MLM** based on our construction of the types of α-iterative sets. Another future research is to see our construction from the viewpoint of recent type-theoretic approaches to ordinals in the context of homotopy type theory [6, 4]. In [6], the authors discuss the conception of ordinals as Brouwer trees, which have several similarities with iterative sets. In [4], the authors discuss a refinement of Aczel's interpretation of **CZF** in MLTT.

[2]https://github.com/takahashi-yt/czf-in-mahlo

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] Tom de Jong, Nicolai Kraus, Fredrik Nordvall Forsberg, and Chuangjie Xu. Set-theoretic and type-theoretic ordinals coincide. *CoRR*, abs/2301.10696, 2023.

[5] Peter Dybjer and Anton Setzer. Induction-recursion and initial algebras. *Ann. Pure Appl. Log.*, 124(1-3):1–47, 2003.

[6] Nicolai Kraus, Fredrik Nordvall Forsberg, and Chuangjie Xu. Type-theoretic approaches to ordinals. *CoRR*, abs/2208.03844, 2022.

[7] Paul Mahlo. Über lineare transfinite Mengen. *Berichte über die Verhandlungen der Königlich Sächsischen Gesellschaft der Wissenschaften zu Leipzig. Mathematisch-Physische Klasse*, 63:187–225, 1911.

[8] Michael Rathjen. Realizing Mahlo set theory in type theory. *Arch. Math. Log.*, 42(1):89–101, 2003.

[9] 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.

[10] Anton Setzer. Extending Martin-Löf type theory by one Mahlo-universe. *Arch. Math. Log.*, 39(3):155–181, 2000.

[11] 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.

Categories as Semicategories with Identities

Joshua Chen, Tom de Jong, Nicolai Kraus, and Stiéphen Pradal

University of Nottingham, Nottingham, UK
{joshua.chen, tom.dejong, nicolai.kraus, stiephen.pradal}@nottingham.ac.uk

Motivation The development of category theory inside type theory has a long history, and many libraries for proof assistants such as Agda or Coq contain results on categories [4, 9, 10, 11, 12, 14]. In a type theory without UIP, in particular in HoTT, the theory of 1-categories is often not applicable for the study of types and more general (i.e., higher) notions of categories are required. For example, the universe of types and functions is adequately described as an $(\infty, 1)$-category. Unsurprisingly, already writing down the definition of such a higher category is involved and a careful approach to organising the huge number of components is needed.[1]

One approach to defining higher categories is to first consider the composition structure (i.e., morphisms, composition, associativity, Mac Lane's pentagon coherence, ...). This leads to a notion of higher semicategory. We then want to describe the higher categories as those higher semicategories that happen to have identities. If we can formulate "having identities" as a propositional property, the higher categories become a subtype of the higher semicategories.

In this talk, we present several different (equivalent) definitions of the property "having identities". Instead of higher categories, we work with a 1-categorical notion of semicategory, a "wild" (untruncated) and a priori ill-behaved concept that generalises both "honest" semicategories (with set-truncated morphism types) and $(\infty, 1)$-semicategories (with all coherences). The fact that this is possible is very fortunate as it simplifies the situation significantly compared to the ∞-categorical setting, but it of course leads to the question in which sense our identity structures are "correct" for $(\infty, 1)$-categories. We discuss this question at the end.

Notions of identities in wild semicategories A wild semicategory is a tuple $(\mathsf{Ob}, \mathsf{hom}, \circ, \alpha)$ where α witnesses associativity. The attribute *wild* indicates that we do not place a truncation condition on the family hom.

Naive identities A direct way to define an identity structure is to ask for a function $\mathsf{id} : \Pi_{x:\mathsf{Ob}} \mathsf{hom}(x, x)$ together with identity laws $\lambda_f : \mathsf{id}_y \circ f = f$ and $\rho_f : f \circ \mathsf{id}_x = f$. Since hom is not required to be a family of sets, this formulation of *having naive identities* is not a proposition and it does not automatically satisfy the coherences that one would expect of an identity in a higher category, such as $\lambda_{\mathsf{id}} = \rho_{\mathsf{id}}$. We write NaId_x for the type of triples $(\mathsf{id}_x, \lambda, \rho)$.

Idempotent equivalences A less direct but more well-behaved definition of an identity structure is to ask for an *idempotent equivalence* on each object ([7]; cf. the *weak units* of [5]). Here, a morphism f is an equivalence if both pre- and post-composition with f is an equivalence of types in the usual (HoTT) sense and we write $\mathsf{eqv}(x, y)$ for the subtype of $\mathsf{hom}(x, y)$ that are equivalences. A morphism $f : \mathsf{hom}(x, x)$ is idempotent if $f \circ f = f$. Clearly, we would expect an identity morphism to be both an equivalence and idempotent, and it turns out that this expectation can be reversed: an idempotent equivalence is always a naive identity in the above sense. This notion is well-behaved since the type $\mathsf{IdemEqv} :\equiv \Pi_{x:\mathsf{Ob}} \Sigma_{i:\mathsf{eqv}(x,x)} (i \circ i = i)$ is a proposition [7].

[1]It is a well-known open question, and one of the major unsolved problems of the field, whether homotopy type theory [13] is expressive enough to formulate the definition of an $(\infty, 1)$-category such that the universe is an instance. The difficulty is to find a way (or determine that there is no way) to encode the infinite number of morphism levels. 2LTT [1] is a setting in which this can be done. The current abstract is *not* on this issue.

Harpaz's identities Following an idea by Harpaz [3], we can ask that there is an equivalence out of each object x, that is, $\Sigma_{y:\mathsf{Ob}}\,\mathsf{eqv}(x,y)$. If we want this to be a proposition, we can truncate (i.e., replace Σ_y by $\exists_y$); this variation is still sufficiently strong to derive a naive identity structure. Alternatively, we can ask for the type of outgoing equivalences (the type of tuples (y,f,e)) to be contractible. It turns out that this version defines *univalent* identities [2]. We write $\mathsf{HarpazId} :\equiv \Pi_{x:\mathsf{Ob}}\exists_{y:\mathsf{Ob}}\,\mathsf{eqv}(x,y)$ and $\mathsf{uHarpazId} :\equiv \Pi_{x:\mathsf{Ob}}\,\mathsf{isContr}(\Sigma_{y:\mathsf{Ob}}\,\mathsf{eqv}(x,y))$.

Identities via (co)slices In category theory, the identity on x is the terminal (resp. initial) object in the slice over (resp. the coslice under) x. Reversing this, we get yet another method to characterise an identity structure in semicategories. Note that wild semicategories are not sufficiently well-behaved to construct slices or coslices as associativity cannot be derived (as explained in e.g. [7]). However, sufficient structure can be constructed to define what it means to be initial or terminal. After unfolding the definition, this leads to the simple definition $\mathsf{SliceId} :\equiv \Pi_{x:\mathsf{Ob}}\|\mathsf{eqv}(x,x)\|$.

Equivalence of the above notions For a semicategory with set-truncated families of morphisms, a naive identity structure is unique if it exists; in other words, NaId is a proposition. This is not the case for a wild semicategory, but we could explicitly truncate to get a proposition $\|\mathsf{NaId}\|$. By combining results from several papers we can then show:

Theorem 1. *For a given wild semicategory, the four types* $\Pi_{x:\mathsf{Ob}}\|\mathsf{NaId}_x\|$, $\mathsf{IdemEqv}$, $\mathsf{HarpazId}$, $\mathsf{SliceId}$ *are equivalent propositions.*

Proof. Three of the types are explicitly constructed to be propositions. In contrast, it is not automatic that $\mathsf{IdemEqv}$ is a proposition: While *being an equivalence* is a proposition, the *type* of equivalences is in general not a proposition, and neither is the statement that a morphism is idempotent. The result was shown by the third-named author in [7] and the strategy is to show that, if an identity-like morphism is given, then every idempotent equivalence has to be equal to it. We refer to the formalisation[2] for the details.

- $\Pi_{x:\mathsf{Ob}}\|\mathsf{NaId}_x\| \leftrightarrow \mathsf{IdemEqv}$ [7]: Naive identities are idempotent equivalences and vice versa.
- $\mathsf{HarpazId} \to \mathsf{IdemEqv}$: This uses an insight of Harpaz [3] in a type-theoretic setting. Given an equivalence $f : \mathsf{hom}(x,y)$, we can apply the inverse of $(f \circ \_)$ to f itself, and the result is an idempotent equivalence.
- Finally, $\mathsf{IdemEqv} \to \mathsf{SliceId} \to \mathsf{HarpazId}$ is easy. □

Discussion One approach to defining $(\infty,1)$-semicategories in a type-theoretic setting is to consider certain type-valued presheaves over the semi-simplex category Δ_+ [1, 2]. Morally, an identity structure corresponds to the maps present in the simplex category Δ but not in Δ_+. Unfortunately, the strategy of defining strict type-valued presheaves via type families only works for direct categories, which Δ is not. Approaches that include an identity structure include the use of a *direct replacement* of Δ [6, 8] or *homotopy coherent nerves* [8]; these structures consist of infinite towers of coherence data.

An $(\infty,1)$-semicategory $\mathcal{C}$ has an underlying wild semicategory $\mathcal{C}_1$. If $\mathcal{C}$ has an identity structure given by an infinite tower of coherence data, then $\mathcal{C}_1$ is trivially equipped with naive identities and thus any of the other discussed identity structures (apart from $\mathsf{uHarpazId}$, which is stronger). We conjecture that the converse holds as well; special cases of this expectation are verified in [2]. This conjecture would give us an easy way to construct the complete tower of coherences by checking any of the very easy conditions discussed above.

[2]**Formalisation** — browsable `html` version: `joshchen.io/agda/semicategories-with-identities/`; Agda source code: `github.com/jaycech3n/semicategories-with-identities`

References

[1] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. Two-level type theory and applications. arXiv[cs.LO]:1705.03307, 2019.

[2] Paolo Capriotti and Nicolai Kraus. Univalent higher categories via complete semi-Segal types. *Proceedings of the ACM on Programming Languages*, 2(POPL'18):44:1–44:29, 2017. Full version available at arXiv:1707.03693.

[3] Yonatan Harpaz. Quasi-unital ∞-categories. *Algebraic & Geometric Topology*, 15(4):2303–2381, 2015. doi:10.2140/agt.2015.15.2303.

[4] Jason Z. S. Hu and Jacques Carette. Agda-Categories: Category theory library for Agda. doi:10.1145/3410272, 2021.

[5] André Joyal and Joachim Kock. Coherence for weak units. *Documenta Mathematica*, 18:71–110, 2013.

[6] Joachim Kock. Weak identity arrows in higher categories. *International Mathematics Research Papers*, 2006:69163, 2006. doi:10.1155/IMRP/2006/69163.

[7] Nicolai Kraus. Internal ∞-categorical models of dependent type theory : Towards 2LTT eating HoTT. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–14, 2021. doi:10.1109/LICS52264.2021.9470667.

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

[9] Amélia Liao, Astra Kolomatskaia, and Reed Mullanix. 1lab. Available at https://1lab.dev/.

[10] The mathlib Community. The Lean mathematical library. *9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CCP'20)*, pages 367–381, 2020. doi:10.1145/3372885.3373824.

[11] Anders Mörtberg, Evan Cavallo, Felix Cherubini, Max Zeuner, Alex Ljungström, Andrea Vezzosi, et al. A standard library for Cubical Agda. Available at https://github.com/agda/cubical, 2018.

[12] Marco Perone et al. Idris category theory. Available at https://github.com/statebox/idris-ct.

[13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book/, 2013.

[14] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath — a computer-checked library of univalent mathematics. Available at http://unimath.org.

New Observations on the Constructive Content of First-Order Completeness Theorems

Hugo Herbelin[1] and Dominik Kirst[2,3]

[1] Université Paris Cité, Inria, CNRS, IRIF, Paris
[2] Saarland University, Germany
[3] Ben-Gurion University, Israel

Abstract

We report on some new observations regarding the constructive reverse mathematics of first-order completeness theorems. When conducted in a constructive type theory such as the calculus of inductive constructions (CIC), many different formulations of completeness can be distinguished and analysed regarding their sufficient and necessary non-constructive assumptions. As the main new result, we identify a principle we dub WLEMS at the intersection of double-negation shift and weak excluded middle, exactly capturing the non-constructivity needed for object-level disjunctions. The observations reported here are part of an ongoing general attempt at a systematic classification of the independent ingredients contributing to the non-constructivity of completeness theorems.

Background Completeness of a logic states that every formula φ semantically entailed by a (possibly infinite) context Γ, denoted by $\Gamma \vDash \varphi$, also admits a syntactical derivation via the rules of a suitable deduction calculus, denoted by $\Gamma \vdash \varphi$. After the discovery that Gödel's completeness theorem of first-order logic [5] relies on Markov's principle (MP) [11], this first-order completeness has been of ongoing interest for the programmes of reverse mathematics [4, 14] and constructive reverse mathematics [9].

Seeking to pin down the exact (non-constructive) assumptions required by the analysed theorems, these programmes are carried out in purposefully weak (constructive) logical systems that allow fine distinctions of logical strength. Among the known results regarding completeness are, besides the mentioned connection to MP, equivalences to the weak König's lemma [15], the weak fan theorem [12], as well as the Boolean prime ideal theorem [6].

Formulations of Completeness Working in CIC [2, 13] and modelling first-order logic in an established way [10], we distinguish the following forms of completeness:

- Completeness: $\forall \varphi, \Gamma.\, \Gamma \vDash \varphi \to \Gamma \vdash \varphi$
- Quasi-Completeness: $\forall \varphi, \Gamma.\, \Gamma \vDash \varphi \to \neg\neg(\Gamma \vdash \varphi)$
- Model Existence: $\forall \varphi, \Gamma.\, \Gamma \nvdash \varphi \to \exists M.\, M \vDash \Gamma \land M \nvDash \varphi$

Usual (Henkin-style) proofs establish model existence first, from which then only quasi-completeness follows constructively, given its additional double negation. Next to these forms of completeness, there are four other dimension contributing to the (non)-constructivity, namely

- The complexity of the context (e.g., finite, decidable, enumerable, arbitrary),
- The cardinality of the signature (e.g., countable, uncountable),
- The syntax fragment (e.g., propositional, minimal, negative, full), and
- The representation of the semantics (e.g., Boolean, decidable, propositional).

In this abstract we discuss the cases of quasi-completeness and model existence for arbitrary contexts over a countable signature, regarding the full syntax (including the critical case of disjunctions) and propositional semantics, which was left open in previous work [8, 3, 7].

Main Observations For the mentioned target formulation of completeness we identify the following principle we call *weak excluded middle shift* (WLEMS):[1]

$$\forall p : \mathbb{N} \to \mathbb{P}.\, \neg\neg(\forall n.\, \neg p\, n \lor \neg\neg p\, n)$$

Note that WLEMS follows both from double-negation shift (DNS),[2] as this would allow to push the outer double negation through the universal quantifier, and from weak excluded middle (WLEM),[3] as this allows the inner classical case distinctions. Both DNS and WLEM give rise to proofs of quasi-completeness themselves, however subsumed by the following main observation:

Theorem 1. *Quasi-completeness (for the said setting) is equivalent to WLEMS.*

Sketch. First assume $\Gamma \vDash \varphi$ and $\Gamma \nvdash \varphi$ for a contradiction. The latter allows to constructively extend Γ to Δ with several closure properties (neglecting the usual treatment of quantifiers):

- Relative Consistency: $\Delta \nvdash \varphi$
- Deductive Closure: $\forall \psi.\, \Delta \vdash \psi \to \psi \in \Delta$
- Stability: $\forall \psi.\, \neg\neg(\psi \in \Delta) \to \psi \in \Delta$
- Quasi-Primeness: $\forall \psi, \psi'.\, \psi \dot\lor \psi' \in \Delta \to \neg\neg(\psi \in \Delta \lor \psi' \in \Delta)$

Combining WLEMS and stability, we can pull the double negation in quasi-primeness to the front and, given the goal to be deriving a contradiction, obtain actual primeness, which is quasi-primeness without any double negations. In a usual completeness proof, primeness is exactly the property needed to verify that the syntactic model M_Δ arising from Δ satisfies both $M_\Delta \vDash \Gamma$ and $M_\Delta \nvDash \varphi$, in contradiction to the assumption $\Gamma \vDash \varphi$.

Conversely given $p : \mathbb{N} \to \mathbb{P}$ with $\neg(\forall n.\, \neg p\, n \lor \neg\neg p\, n)$ for a contradiction, we consider

$$\Gamma := \{P_n \dot\lor \dot\neg P_n \mid n : \mathbb{N}\} \cup \{P_n \mid p\, n\} \cup \{\dot\neg P_n \mid \neg p\, n\}$$

using countably many propositional variables P_n. Applying quasi completeness for $\varphi := \dot\bot$, we are left to show that $\Gamma \vDash \dot\bot$ and that Γ is consistent. The latter is possible using a suitable model and soundness, the former boils down to assuming a model M with $M \vDash \Gamma$ and then showing $\forall n.\, \neg p\, n \lor \neg\neg p\, n$ by inspecting the choices $M \vDash P_n \dot\lor \dot\neg P_n$ made by the model. □

Complementing the previous equivalence, we also observe:

Theorem 2. *Model existence (for the said setting) is equivalent to WLEM.*

Sketch. WLEM is enough to show that stable quasi-prime theories are actually prime, yielding model existence as above. Conversely given $p : \mathbb{P}$, model existence for the consistent context $\Gamma := \{P_0 \dot\lor \dot\neg P_0\} \cup \{P_0 \mid p\} \cup \{\dot\neg P_0 \mid \neg p\}$ yields the desired case distinction $\neg p \lor \neg\neg p$. □

Outlook First, although presented here for classical first-order logic, we expect that the same results hold for intuitionistic first-order logic. Actually, we suspect that the characterisation of disjunction using WLEMS and WLEM is universal enough to apply also to other logics like modal logic or bi-intuitionistic logic. Secondly, it would be desirable to develop an abstract framework for completeness proofs, orthogonalising the different dimensions of non-constructivity and generalising over the concrete specifics of the analysed logic. Thirdly, we want to investigate the exact correlation of WLEMS and formulations of the weak fan theorem, especially regarding Berger's decomposition of the latter [1].

[1]Equivalent to *disjunctive double-negation shift*: $\forall pq : \mathbb{N} \to \mathbb{P}.\, (\forall n.\, \neg\neg(\neg p\, n \lor \neg q\, n)) \to \neg\neg(\forall n.\, \neg p\, n \lor \neg q\, n)$ and mentioned in more general form as M° quantifying over arbitrary domains by Umezawa [16].

[2]$\forall X.\, \forall p : X \to \mathbb{P}.\, (\forall x.\, \neg\neg p\, x) \to \neg\neg(\forall x.\, p\, x)$

[3]$\forall p : \mathbb{P}.\, \neg p \lor \neg\neg p$

References

[1] Josef Berger. A decomposition of Brouwer's fan theorem. *Journal of Logic and Analysis*, 1, 2009.

[2] Thierry Coquand. *The calculus of constructions.* PhD thesis, INRIA, 1986.

[3] Yannick Forster, Dominik Kirst, and Dominik Wehr. Completeness theorems for first-order logic analysed in constructive type theory. In *International Symposium on Logical Foundations of Computer Science*. Springer, 2020.

[4] Harvey Friedman. Some systems of second order arithmetic and their use. In *Proceedings of the international congress of mathematicians (Vancouver, BC, 1974)*, volume 1, pages 235–242. Citeseer, 1975.

[5] Kurt Gödel. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik*, 37:349–360, 1930. URL: https://zbmath.org/?q=an%3A56.0046.04.

[6] Leon Henkin. Metamathematical theorems equivalent to the prime ideal theorem for boolean algebras. *Bulletin AMS*, 60:387–388, 1954.

[7] Hugo Herbelin. Computing with Gödel's completeness theorem: Weak fan theorem, Markov's principle and double negation shift in action. http://pauillac.inria.fr/~herbelin/talks/chocola22.pdf, 2022.

[8] Hugo Herbelin and Danko Ilik. An analysis of the constructive content of Henkin's proof of Gödel's completeness theorem. 2016.

[9] Hajime Ishihara. Reverse Mathematics in Bishop's Constructive Mathematics. *Philosophia Scientae*, pages 43–59, 2006. doi:10.4000/philosophiascientiae.406.

[10] Dominik Kirst, Johannes Hostert, Andrej Dudenhefner, Yannick Forster, Marc Hermes, Mark Koch, Dominique Larchey-Wendling, Niklas Mück, Benjamin Peters, Gert Smolka, and Dominik Wehr. A Coq library for mechanised first-order logic. In *Coq Workshop*, 2022.

[11] Georg Kreisel. On weak completeness of intuitionistic predicate logic. *The Journal of Symbolic Logic*, 27(2):139–158, 1962.

[12] Victor N Krivtsov. Semantical completeness of first-order predicate logic and the weak fan theorem. *Studia Logica*, 103(3):623–638, 2015.

[13] Christine Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, pages 328–345. Springer, 1993.

[14] Stephen G. Simpson. Reverse mathematics. In *Proc. Symposia Pure Math*, volume 42, pages 461–471, 1985.

[15] Stephen G. Simpson. *Subsystems of second order arithmetic*, volume 1. Cambridge University Press, 2009.

[16] Toshio Umezawa. On logics intermediate between intuitionistic and classical predicate logic. *The Journal of Symbolic Logic*, 24(2):141–153, 1959.

9

Session 11: Automation in computer-assisted reasoning

Embedding Differential Temporal Dynamic Logic in PVS

Lauren White[1], Laura Titolo[2], and J. Tanner Slagel[1]

[1] NASA Langley Research Center, USA
{lauren.m.white,j.tanner.slagel}@nasa.gov
[2] National Institute of Aerospace
laura.titolo@nianet.org

1 Introduction

Differential dynamic logic (dL) [11, 12, 13, 14] is a formal framework to specify and reason about hybrid programs (HPs). The core of dL is a proof calculus that contains a collection of axioms and rules for the rigorous verification of properties of HPs. This calculus is implemented in the KeYmaera X[1] theorem prover which has been used for formal verification of several cyber-physical systems [4, 8, 6, 2, 1, 3, 7, 9]. Recently, dL has been embedded within the theorem prover Prototype Verification System (PVS) [10] resulting in the tool Plaidypvs[2]. The integration of dL into PVS expands its expressive power; user defined functions, such as trigonometric and other transcendental functions, can be used inside the dL framework, and meta-reasoning about HPs can be performed, including reasoning about entire classes of HPs, specified using dependent types in PVS.

One limitation of dL, KeYmaera X, and Plaidypvs is that they can only reason on the input/output semantics of an HP. Nevertheless, it is often the case that the correctness of an HP depends on the intermediate states that it can reach during its executions. For example, guaranteeing the position of an aircraft stays within a geofenced region. The differential temporal dynamic logic (dTL^2) has been introduced in [5] to extend dL with temporal logic operators and reason about all the states reachable during the execution of an HP.

This paper presents a work in progress focusing on embedding dTL^2 in PVS as an extension of Plaidypvs. Plaidypvs is expanded with the formalization of a trace semantics for HPs, the definition of the LTL temporal operators *eventually* and *globally*, and the implementation of the proof calculus for dTL^2. This new embedding has the same capabilities as Plaidypvs, which allows user defined functions and meta-reasoning of properties of HPs.

2 Embedding dTL^2 in PVS

HPs combine discrete programs and continuous evolutions and are defined by the syntax

$$\alpha ::= \ x := \theta \mid x' := \theta \,\&\, P \mid ?P \mid x := * \,\&\, P \mid \alpha;\alpha \mid \alpha \cup \alpha \mid \alpha^*$$

where x and x' are variables, θ is a real expression, and P is a Boolean expression. The statement $x := \theta$ denotes a discrete assignment, $x' := \theta \,\&\, P$ models the continuous first order differential equation defined by θ that satisfies P, $?P$ tests if P is satisfied, and $x := * \,\&\, P$ assigns to x an arbitrary real value r such that $P(r)$ holds. HPs can be combined through sequential execution ($\alpha_1;\alpha_2$), nondeterministic choice ($\alpha_1 \cup \alpha_2$), and nondeterministic finite repetition (α_1^*).

[1] https://keymaerax.org
[2] Plaidypvs is part of the NASA PVS library available at https://github.com/nasa/pvslib/tree/master/dL.

Real and Boolean expressions are shallowly embedded in PVS. Given a state (or environment) in $\mathbb{S}$ which maps variables into $\mathbb{R}$, a real expression has type $[\mathbb{S} \to \mathbb{R}]$, while a Boolean expression has type $[\mathbb{S} \to \mathbb{B}]$ where $\mathbb{B}$ is the Boolean domain. This type of embedding is more general and easily extendable than the deep embedding where each operator must be defined with a dedicated datatype. Additionally, it allows interpretation of the operators directly in the logic of PVS, facilitating the task of writing the proofs.

The semantics of an HP is defined as a set of traces. In PVS, a trace is defined as a function $\sigma : \mathbb{N} \times \mathbb{R} \to \mathbb{S}$. A state $\sigma(i, r) \in \mathbb{S}$ denotes the state in trace σ occurring at the discrete step i and at time r. In the following, let σ_i denote $\sigma(i, 0)$ which is defined on the interval $[0, 0]$ and is used to model discrete steps. The trace semantics is defined as a relation $\tau(\alpha, t)$ that holds when a trace t belongs to the semantics of an HP α. For instance, it holds that $\tau(x := \theta, \sigma_0\sigma_1)$ when $\sigma_1 = \sigma_0[x/\theta]$ and $\tau(x' := \theta \,\&\, P, \sigma')$ where σ' is a state flow of order one solution of θ defined on $[0, r]$ such that for all $t \in [0, r]$, $\sigma(t)$ satisfies P.

There are two kinds of formulas in dTL^2: *state formulas* (ϕ) that are interpreted over a single state and trace formulas (π) that are interpreted over a trace:

$$\begin{aligned}
&\phi ::= \theta_1 \geq \theta_2 \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \forall x\ \phi \mid \exists x\ \phi \mid [\alpha]\pi \mid \langle\alpha\rangle\pi \\
&\pi ::= \phi \mid \neg\pi \mid \Box\pi \mid \Diamond\pi
\end{aligned}$$

where θ_1 and θ_2 are real expressions. The run quantification statement $[\alpha]P$ asserts that every run of α ends by satisfying P. Similarly, $\langle\alpha\rangle P$ asserts that there exists a run of α where the final state satisfies P. The operators $\Box$, globally, and $\Diamond$, eventually, are defined in the typical way according to LTL with the restriction that they can be nested at most twice, i.e., the only combinations allowed are $\Box\Diamond$ and $\Diamond\Box$. While this can look like a stringent limitation, the combination of the LTL temporal operators with the dL run quantification allows reasoning on the reachable states of different computational paths.

Each proof rule of dTL^2 presented in [5] is specified as a lemma and proven correct in PVS. The non-temporal rules that reason on state formulas are inherited from Plaidypvs. For instance, the rules for proving a globally statement on all the runs of an assignment and a sequential composition are the following.

$$\frac{[x := \theta](\phi)}{[x := \theta](\Box\phi)} \qquad \frac{[\alpha_1][\alpha_2](\Box\phi)}{[\alpha_1; \alpha_2](\Box\phi)}$$

A lemma that encodes a desired rule can be instantiated in the PVS proof environment to prove a given property for an HP. In the future, proof strategies automating this process will be developed.

3 Conclusions

This extended abstract presents a work in progress for the implementation of the dTL^2 logic in PVS. Upon the completion of this work, the formalization of dTL^2 will be added to the Plaidypvs tool and made available in the NASA PVS library. The combination of LTL operators with dL allows for reasoning about the intermediate states on different computational paths of an HP, enabling an interesting fragment of CTL*. The dTL^2 embedding in PVS is implemented as a mixture of deep and shallow embeddings enabling user defined functions, and meta-reasoning about HPs using dependent types in PVS. To the best of the authors' knowledge this is the first implementation of dTL^2.

References

[1] B. Bohrer, Y.K. Tan, S. Mitsch, A. Sogokon, and A. Platzer. A formal safety net for waypoint following in ground robots. *IEEE Robotics and Automation Letters*, 4(3):2910–2917, 2019.

[2] R. Cleaveland, S. Mitsch, and A. Platzer. Formally verified next-generation airborne collision avoidance games in ACAS X. *ACM Trans. Embed. Comput. Syst.*, 22(1):1–30, 2023.

[3] N. Fulton and A. Platzer. Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, pages 6485–6492. AAAI Press, 2018.

[4] J. Jeannin, K. Ghorbal, Y. Kouskoulas, A. Schmidt, R. Gardner, S. Mitsch, and A. Platzer. A formally verified hybrid system for safe advisories in the next-generation airborne collision avoidance system. *International Journal on Software Tools for Technology Transfer*, 19(6):717–741, 2017.

[5] J. Jeannin and A. Platzer. dtl2: Differential temporal dynamic logic with nested temporalities for hybrid systems. In *Proceedings of the 7th International Joint Conference on Automated Reasoning (IJCAR 2014)*, volume 8562 of *LNCS*, pages 292–306. Springer, 2014.

[6] A. Kabra, S. Mitsch, and A. Platzer. Verified train controllers for the federal railroad administration train kinematics model: Balancing competing brake and track forces. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4409–4420, 2022.

[7] S. Mitsch, M. Gario, C. J. Budnik, M. Golm, and A. Platzer. Formal verification of train control with air pressure brakes. In *Proceedings of the 2nd International Conference Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification (RSSRail 2017)*, volume 10598 of *LNCS*, pages 173–191. Springer, 2017.

[8] S. Mitsch, K. Ghorbal, D. Vogelbacher, and A. Platzer. Formal verification of obstacle avoidance and navigation of ground robots. *International Journal of Robotics Research*, 36(12):1312–1340, 2017.

[9] A. Müller, S. Mitsch, W. Retschitzegger, W. Schwinger, and A. Platzer. Change and delay contracts for hybrid system component verification. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE 2017)*, volume 10202 of *LNCS*, pages 134–151. Springer, 2017.

[10] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE 1992)*, pages 748–752. Springer, 1992.

[11] A. Platzer. Differential dynamic logic for verifying parametric hybrid systems. In *Proceedings of 16th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2007)*, volume 4548 of *LNCS*, pages 216–232. Springer, 2007.

[12] A. Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189, 2008.

[13] A. Platzer. A complete uniform substitution calculus for differential dynamic logic. *Journal of Automated Reasoning*, 59(2):219–265, 2017.

[14] A. Platzer. *Logical Foundations of Cyber-Physical Systems.* Springer, 2018.

A record expansion translation for Coq

Kazuhiko Sakaguchi

Inria, France

Abstract

In proof assistants based on dependently-typed languages, record types have been used as interfaces for abstraction bundling some objects, including mathematical structures. However, these record types introduce some indirections, namely, constructions and destructions of records, which can be a source of efficiency and intelligibility issues in translations such as extraction and parametricity translation. We propose a translation from Coq to Coq that expands record types and eliminates these indirections.

Structures as records In proof assistants based on dependently-typed languages, record types [6] have been used as interfaces for abstraction that bundle some objects. For example, the following record type written in Coq [13], which is a simplified version of the `eqType` structure in the MathComp library [17], represents a type `eq_type` equipped with a comparison function `eq_op` and a proof `eqP` that `eq_op x y` is `true` if and only if `x` and `y` are propositionally equal.

```
Structure eqType := EqPack {
  eq_sort :> Type;
  eq_op : eq_sort -> eq_sort -> bool;
  eqP : forall x y : eq_sort, reflect (x = y) (eq_op x y) }.
```

We can define a function that works generically for any instance of the above record type. As an example, we define `mem_seq` that determines whether a list includes a given element by using `eq_op` for comparison, and `undup` that eliminates duplications in a given list.

```
Definition mem_seq (T : eqType) (x : eq_sort T) :=
  fix rec (ys : seq (eq_sort T)) : bool :=
    if ys is y :: ys then eq_op x y || rec ys else false.

Definition undup (T : eqType) :=
  fix rec (xs : seq (eq_sort T)) : seq (eq_sort T) :=
    if xs is x :: xs then
        let xs' := rec xs in if mem_seq x xs' then xs' else x :: xs'
    else [::].
```

where `eq_sort`, intentionally made explicit, can be omitted thanks to the implicit coercion mechanism [14].

This approach of abstraction, in combination with mechanisms to automatically infer record instances [5, 9, 15, 16], has been extensively used to define and reason about mathematical structures such as groups, rings, and fields, that may form a complex inheritance hierarchy [1, 2, 10, 18].

Indirections The use of record types as interfaces for abstraction introduces some indirections in raw terms, namely, construction and destruction of records. Furthermore, an instance of a richer structure, *e.g.*, a field, sometimes has to be *repackaged* as an instance of a poorer structure, *e.g.*, a ring, to deal with structure inheritance, *e.g.*, the fact that any field form a ring [1, Section 2.4]. While these indirections are necessary ingredients to enable structure inference and most of them are made implicit in the user-facing syntax, they can be a problem in translations that operate on raw terms, such as program extraction [4] and parametricity translation [3]. For example, `mem_seq` and `undup` above are extracted to the following OCaml program.

```
type eq_sort = Obj.t

(** val mem_seq : eqType -> eq_sort -> eq_sort list -> bool **)
let rec mem_seq t x = function
| [] -> false
| y :: ys0 -> (||) (t.eq_op x y) (mem_seq t x ys0)

(** val undup : eqType -> eq_sort list -> eq_sort list **)
let rec undup t = function
| [] -> []
| x :: xs0 -> let xs' = undup t xs0 in if mem_seq t x xs' then xs' else x :: xs'
```

Since the type of elements of the list is bundled in the first parameter of type `eqType`, it is translated to `Obj.t`, which has to be coerced by `Obj.magic` when these definitions are specialized to a concrete `eqType` instance. Therefore, they do not have polymorphic types one may expect. Moreover, `mem_seq` has to access to the comparison function through the record projection `eq_op`, which is a source of inefficiencies particularly in the case involving deeply nested records and inheritance [7, Section 5.1].

Record expansion translation We propose a translation from Coq to Coq that eliminates the indirection issues by expanding a record type to its fields. For example, `T` of type `eqType` in our example can be expanded to the carrier type `eq_sort`, the function `eq_op`, and the proof `eqP`. Record projections applied to an expanded record can then be reduced to the corresponding field. Since the proof `eqP` is unused in `mem_seq` and `undup`, it can be removed.

```
Definition mem_seq' (T : Type) (eq_op : T -> T -> bool) (x : T) :=
  fix rec (ys : seq T) : bool :=
     if ys is y :: ys then eq_op x y || rec ys else false.

Definition undup' (T : Type) (eq_op : T -> T -> bool) :=
  fix rec (xs : seq T) : seq T :=
    if xs is x :: xs then
      let xs' := rec xs in if mem_seq' eq_op x xs' then xs' else x :: xs'
    else [::].
```

In general, an abstraction over a record type has to be expanded to abstractions over its fields. On the other hand, a constant that returns a record instance has to be expanded to several constants corresponding to each field.

As a result, the extracted OCaml functions now have the expected polymorphic type, and the use of the record projection `eq_op` disappears.

```
(** val mem_seq' : ('a1 -> 'a1 -> bool) -> 'a1 -> 'a1 list -> bool **)
let rec mem_seq' eq_op x = function
| [] -> false
| y :: ys0 -> (||) (eq_op x y) (mem_seq' eq_op x ys0)

(** val undup' : ('a1 -> 'a1 -> bool) -> 'a1 list -> 'a1 list **)
let rec undup' eq_op = function
| [] -> []
| x :: xs0 -> let xs' = undup' eq_op xs0 in if mem_seq' eq_op x xs' then xs' else x :: xs'
```

We are currently working on an implementation of this translation in Coq-Elpi [11]. We plan to produce proofs certifying each translation since the translation itself will not be verified, while its reimplementation in MetaCoq [8] would allow us to verify the translation itself. In many cases, such a proof, *e.g.*, that the translated definition `undup'` is equal to `undup`, can be done by reflexivity. However, it is crucial that the record to expand does not appear as an argument of fixpoint functions in the definition.

```
Lemma undupE T eq_op eqP : @undup (@EqPack T eq_op eqP) = @undup' T eq_op.
Proof. reflexivity. Qed.
```

References

[1] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009.

[2] Herman Geuvers, Randy Pollack, Freek Wiedijk, and Jan Zwanenburg. A constructive algebraic hierarchy in Coq. *J. Symb. Comput.*, 34(4):271–286, 2002.

[3] 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, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 381–395. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[4] Pierre Letouzey. *Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq. (Certified functional programming : Program extraction within Coq proof assistant)*. PhD thesis, University of Paris-Sud, Orsay, France, 2004.

[5] Assia Mahboubi and Enrico Tassi. Canonical structures for the working Coq user. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2013.

[6] Robert Pollack. Dependently typed records in type theory. *Formal Aspects Comput.*, 13(3-5):386–402, 2002.

[7] Kazuhiko Sakaguchi. Program extraction for mutable arrays. *Sci. Comput. Program.*, 191:102372, 2020.

[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] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2008.

[10] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Math. Struct. Comput. Sci.*, 21(4):795–825, 2011.

[11] Enrico Tassi. Coq-Elpi, 2017–. Accessed: 2023-03-14.

[12] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2022. the PDF version with numbered sections is available at https://doi.org/10.5281/zenodo.7313584.

[13] The Coq Development Team. *Section 2.1.8 "Record types"*. In [12], 2022.

[14] The Coq Development Team. *Section 2.2.6 "Implicit Coercions"*. In [12], 2022.

[15] The Coq Development Team. *Section 2.2.7 "Typeclasses"*. In [12], 2022.

[16] The Coq Development Team. *Section 2.2.8 "Canonical Structures"*. In [12], 2022.

[17] The Mathematical Components project. The mathematical components repository, 2015–. Accessed: 2023-03-14.

[18] The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 367–381. ACM, 2020.

A graphical interface for diagrammatic proofs in proof assistants

Luc Chabassier and Bruno Barras

Université Paris-Saclay, INRIA project Deducteam,
Laboratoire Méthodes Formelles, ENS Paris-Saclay

Introduction Category Theory is a very active domain in both computer science and mathematics research, with applications from algebraic geometry to programming languages design. Many attempts to formalize parts of this field in proof assistants have been made [1, 5, 6, 11]. They usually suffer from common drawbacks, two of which are :

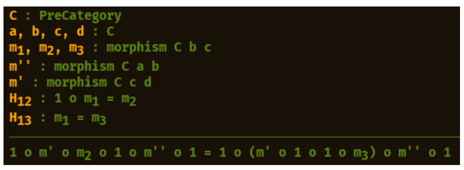
- Reasoning is often done modulo associativity and unitality (and more generally modulo structural laws that depends on the specific categories of interest). As of now, most tactics do not handle such properties, resulting in user to have to do a lot of bureaucratic work to make sure the parentheses are at the right place before rewriting.
- One of the aspects that made category theory so attractive is that it enabled reasoning about equational systems using an intuitive graphical representation based on graphs, usually called *categorical diagrams*. The main idea is to represent morphisms as arrows between their domains and codomains, laid out on the plane. Equalities are then faces between parallel paths on the graphs. Those faces are called commutative, and one can reason about the equalities using visually intuitive rules. This is fundamentally hard to replicate in a text-based proof assistant.

We propose an attempt to alleviate those two, particularly by focusing on the second point. In practice, multiple kinds of graphical calculi exists for specific kind of categories, such as *string diagrams* for *monoidal categories*. This work restrict itself to categorical diagrams.

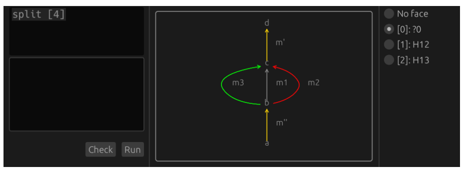
The presentation as graphs has the advantage of factoring out the associativity, since composition are now path in the graph. Our approach thus mostly abstracts away associativity and unitality questions.

Approach We developed a Coq plugin that, from a given goal, constructs a graph that represent the proof state from a categorical point of view. It constructs a graph were each object of a category is a node, and each morphism an edge. Equalities between morphism becomes *faces* in the graph. See figure 1 for an example, with the sides of the goal highlighted.

To represent remaining goals, the terms can include *holes*. These representation breaks the asymmetry between hypothesis and goals, since there only are terms, potentially with holes in them (this is similar to existential variables in Coq).



```
C : PreCategory
a, b, c, d : C
m₁, m₂, m₃ : morphism C b c
m'' : morphism C a b
m' : morphism C c d
H₁₂ : 1 o m₁ = m₂
H₁₃ : m₁ = m₃
________________________________________
1 o m' o m₂ o 1 o m'' o 1 = 1 o (m' o 1 o 1 o m₃) o m'' o 1
```



```
split [4]
```

Figure 1: The Coq proof state and the corresponding graph

From then, the graph can be manipulated graphically or using a language created specifically to manipulate diagrams. Through the manipulations, holes may be filled, or partially filled. When closing the interface, the terms filling the holes are sent to Coq, and their holes become new goals. It is not necessary to conclude the proof in the interface, one can go back and forth between it and Coq, depending on what is the most convenient tool for parts of the proof.

Mathematical papers do not usually bother detailing the last steps of diagrammatic proofs, that only includes formal manipulations. To simulate this, a small solver has been implemented, that can automate some of the trivial diagrammatic reasoning. It works by enumerating paths up to a certain length, and propagating equalities using an union-find.

Lemmas If the interface only allowed structural manipulation of diagrams, its might not be useful enough. However, we developed a system to use lemmas that are *graphical enough* directly from the interface. A lemma is *graphical enough* if it only quantifies on categories, objects, functors, morphisms or equality, and its conclusion is one of those five. In that case, following a procedure similar to the construction of the graph from the context, a graph is constructed from the lemma, using holes for all the terms it quantify universally upon, and skolemizing the existential quantifiers. The term of the conclusion is then the lemma applied to the holes created.

From the interface, the user can display the graph of any lemma, and construct a partial graph matching between the graph of a lemma and the goal. When matching two terms, they are unified. After the unifications have succeeded, applying the lemma consist of taking the pushout of the partial matching. We believe such an operation generalize backward reasoning, forward reasoning, and cuts, while being quite intuitive and graphical in nature.

Architecture This interface is actually split in two programs. This interface itself, along with the solver, and all the implementations of the diagram manipulation, are implemented in rust in a standalone program. This program has no knowledge specific to Coq or even to type theoretic proof assistants. It operates on an abstract representation of terms.

The role of constructing the representation from the concrete terms is left to the Coq plugin, written in OCaml. The plugin is also responsible for building concrete terms from abstract instructions, and constructing the graph from the proof state.

Since most of the logic is fully independent from Coq, it should be quite easy implement plugins for other proof assistants, which also communicates with the interface though the same protocol. Once the latter is a bit more stabilised, we hope to document and version it so that other people can write their own plugin using our interface.

The code source can be found on GitHub[1].

Future work To fully support the workflow of a category theorist, there must be a way to represent and manipulate subdiagrams with structure, like for example isomorphims, pushouts, or pullbacks. Since making an inventory of such structure would be futile, there must be a way to specify them. We are looking into generalised sketches[9] as an inspiration for a formalism to talk about such structures in an abstract way.

Similar tools Standalone proof assistants specific for specific category theories, often with with some graphical calculus, include Globular[3], rzk[7] and graph-editor[8]. Notable integrations of graphical interaction in text based proof assistants include Actema[4] and lean widgets[2], the latter being used to implement diagrams visualisation[10].

[1]`https://github.com/dwarfmaster/commutative-diagrams`

References

[1] Benedikt Ahrens, Chris Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Mathematical Structures in Computer Science*, 25(5):1010–1039, June 2015. arXiv: 1303.0584.

[2] Edward Ayers. A Tool for Producing Verified, Explainable Proofs. September 2021. Publisher: Apollo - University of Cambridge Repository.

[3] Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. Globular: an online proof assistant for higher-dimensional rewriting. In *Leibniz International Proceedings in Informatics*, volume 52, pages 34:1–34:11, 2016.

[4] Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. A drag-and-drop proof tactic. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 197–209, Philadelphia PA USA, January 2022. ACM.

[5] Jason Gross, Adam Chlipala, and David I. Spivak. Experience Implementing a Performant Category-Theory Library in Coq. *arXiv:1401.7694 [cs, math]*, April 2014. arXiv: 1401.7694.

[6] Jason Z. S. Hu and Jacques Carette. Formalizing of Category Theory in Agda. *arXiv:2005.07059 [cs]*, January 2021. arXiv: 2005.07059.

[7] Nikolai Kudasov. An experimental proof assistant for synthetic ∞-categories. `https://github.com/fizruk/rzk`, 2023.

[8] Ambroise Lafont. A vi-inspired diagram editor. `https://amblafont.github.io/graph-editor/index.html`, 2023.

[9] Mihály Makkai and Robert Paré. *Accessible categories: the foundations of categorical model theory.* American Mathematical Society, Providence, R.I, online-ausg edition, 1989. OCLC: 1030355228.

[10] Wojciech Nawrocki. A commutative diagram widget. `https://github.com/leanprover-community/mathlib4/pull/3583`, 2023.

[11] Eugene W. Stark. Category Theory with Adjunctions and Limits. *Archive of Formal Proofs*, June 2016.

10

Session 12: Applications of type theory

Profinite λ-terms and parametricity

Vincent Moreau

Université Paris Cité, IRIF, Inria Paris

The aim of this work is to combine profinite methods and models of the λ-calculus to obtain a notion of *profinite λ-term* which, we show, lives in perfect harmony with the principles of Reynolds parametricity. This is joint work with Sam van Gool and Paul-André Melliès [2].

Languages of finite words and profiniteness. Automata theory has a central role in theoretical computer science. In its most basic form, it deals with regular languages of finite words. If M is a finite monoid and $\varphi : \Sigma^* \to M$ is a monoid homomorphism, then each subset S of M induces the regular language

$$L_S \quad := \quad \{w \in \Sigma^* \mid \varphi(w) \in S\} ,$$

that is the set of words which, when interpreted in the monoid M with the morphism φ, yield an element of S. We recover all the regular languages in this way:

$$\mathrm{Reg}\langle\Sigma\rangle \quad = \quad \{L_S \mid M \text{ a finite monoid, } S \subseteq M\} .$$

Two finite words can be given a distance measuring the minimal cardinality of a finite monoid in which their behaviors are different. The monoid of finite words Σ^* can then be completed into a topological monoid $\widehat{\Sigma^*}$ called the free profinite monoid. Its points, known as profinite words, provide a way to speak about limiting behavior of finite words with respect to finite monoids [4].

Regular languages are closed under union, intersection and complement, which means that the set $\mathrm{Reg}\langle\Sigma\rangle$, ordered under the inclusion, is a Boolean algebra. By Stone duality, it has an associated space of ultrafilters which is in fact homeomorphic to $\widehat{\Sigma^*}$. The monoid structure on $\widehat{\Sigma^*}$ can be seen as the dual of residual operations on $\mathrm{Reg}\langle\Sigma\rangle$, see [1]. In summary,

$$\widehat{\Sigma^*} \quad \text{is the Stone dual of} \quad \mathrm{Reg}\langle\Sigma\rangle . \tag{1}$$

From words to λ-terms: the Church encoding. We consider the simply-typed λ-calculus with one base type $\mathtt{o}$. For any simple type A, we denote by $\Lambda\langle A\rangle$ the set of closed λ-terms of type A, taken modulo $\beta\eta$-conversion. To any finite alphabet Σ, we associate the simple type

$$\mathrm{Church}_\Sigma \quad := \quad \underbrace{(\mathtt{o} \to \mathtt{o}) \to \ldots \to (\mathtt{o} \to \mathtt{o})}_{|\Sigma| \text{ times}} \to \mathtt{o} \to \mathtt{o}$$

and we encode finite words over $\Sigma = \{a_1, \ldots, a_n\}$ as terms of this type in the following way:

$$w = a_{w_1} \ldots a_{w_k} \quad \text{is encoded as} \quad \lambda(a_1 : \mathtt{o} \to \mathtt{o}) \ldots \lambda(a_n : \mathtt{o} \to \mathtt{o})\lambda(c : \mathtt{o}).\, a_{w_k}(\ldots(a_{w_1} c)) .$$

We use the finite standard model of the simply-typed λ-calculus, that is we interpret it in the cartesian closed category **FinSet**. This means that, for any simple type A and finite set Q, we obtain a finite set $[\![A]\!]_Q$ and a function

$$[\![-]\!]_Q \quad : \quad \Lambda\langle A\rangle \longrightarrow [\![A]\!]_Q .$$

In particular, a word $w \in \Sigma^*$ encoded as a simply-typed λ-term of type Church_Σ will be interpreted as a function

$$[\![w]\!]_Q \quad \in \quad (Q \Rightarrow Q) \Rightarrow \ldots \Rightarrow (Q \Rightarrow Q) \Rightarrow Q \Rightarrow Q$$

taking as inputs a deterministic transition function for each letter of the alphabet Σ and an initial state and giving as output the state the automaton arrives at after reading the word w. This shows that the semantics in **FinSet** generalize at any type the usual notion of run into an automaton; see also [3].

Recognizable languages of λ-terms. In analogy with the monoid case, we define regular languages of λ-terms as sets of λ-terms interpreted as certain semantic elements. Following [6], for any simple type A, finite set Q and subset F of $[\![A]\!]_Q$, we define the language L_F as

$$L_F \quad := \quad \{M \in \Lambda\langle A\rangle \mid [\![M]\!]_Q \in F\} \ .$$

We can therefore define the set $\mathrm{Reg}\langle A\rangle$ of all regular languages of λ-terms of type A as

$$\mathrm{Reg}\langle A\rangle \quad := \quad \{L_F \mid Q \text{ a finite set, } F \subseteq [\![A]\!]_Q\} \ .$$

Using logical relations, we can prove again that $\mathrm{Reg}\langle A\rangle$ is a Boolean algebra and then, in analogy with (1), we *define* the space $\widehat{\Lambda}\langle A\rangle$ of profinite λ-terms of type A such that

$$\widehat{\Lambda}\langle A\rangle \quad \text{is the Stone dual of} \quad \mathrm{Reg}\langle A\rangle \ . \tag{2}$$

There is a natural map $\Lambda\langle A\rangle \to \widehat{\Lambda}\langle A\rangle$, which we prove is injective using [7].

Profinite λ-terms and parametricity. As **FinSet** is a cartesian closed category, we can use logical relations. For any relation $R \subseteq P \times Q$, we have a relation $[\![A]\!]_R \subseteq [\![A]\!]_P \times [\![A]\!]_Q$. Following [5], we say that a family θ of elements $\theta_Q \in [\![A]\!]_Q$, where Q ranges over finite sets, is parametric if for any relation $R \subseteq P \times Q$, the elements θ_P and θ_Q are related by $[\![A]\!]_R$. We denote by $\mathrm{Para}\langle A\rangle$ the set of all parametric families associated to a simple type A.

Parametricity can be thought of as the notion of naturality, adapted to the higher-order setting. The fundamental lemma of λ-calculus states that the interpretation of a λ-term is a parametric family. Each profinite λ-term can be thought as a family θ verifying certain properties, see Definition 9 in [2]. We first show the more general result that profinite λ-terms are in particular parametric.

Theorem 1. *For any simple type A, we have $\widehat{\Lambda}\langle A\rangle \subseteq \mathrm{Para}\langle A\rangle$ i.e. for any profinite λ-term θ of type A and relation $R \subseteq P \times Q$, the points θ_P and θ_Q are related by $[\![A]\!]_R$.*

Our main contribution is a parametricity theorem for Church types, which amounts to saying that any parametric family of type Church_Σ is the interpretation of a profinite λ-term.

Theorem 2. *For any finite set Σ, we have $\widehat{\Lambda}\langle \mathrm{Church}_\Sigma\rangle = \mathrm{Para}\langle \mathrm{Church}_\Sigma\rangle$.*

As future work, we would like to investigate the status of this equality for other higher-order types. We are also interested in studying the possibly different notions of profinite λ-term that we obtain when using other cartesian closed categories as models.

References

[1] Mai Gehrke, Serge Grigorieff, and Jean-Éric Pin. Duality and equational theory of regular languages. In L. Aceto and al., editors, *ICALP 2008, Part II*, volume 5126 of *Lecture Notes in Computer Science*, pages 246–257, Berlin, 2008. Springer.

[2] Sam van Gool, Paul-André Melliès, and Vincent Moreau. Profinite lambda-terms and parametricity. To appear in the Proceedings of the 39th International Conference on Mathematical Foundations of Programming Semantics, 2023.

[3] Paul-André Melliès. Higher-order parity automata. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '17. IEEE Press, 2017.

[4] Jean-Éric Pin. Profinite Methods in Automata Theory. In Susanne Albers and Jean-Yves Marion, editors, *26th International Symposium on Theoretical Aspects of Computer Science STACS 2009*, Proceedings of the 26th Annual Symposium on the Theoretical Aspects of Computer Science, pages 31–50, Freiburg, Germany, February 2009. IBFI Schloss Dagstuhl.

[5] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523. North-Holland/IFIP, 1983.

[6] Sylvain Salvati. Recognizability in the simply typed lambda-calculus. In *Logic, Language, Information and Computation*, pages 48–60. Springer Berlin Heidelberg.

[7] Richard Statman and Gilles Dowek. On Statman's finite completeness theorem. Technical report, Carnegie Mellon University, 1992. CMU-CS-92-152.

Finite Combinatory Logic extended by a Boolean Query Language for Composition Synthesis

Andrej Dudenhefner, Felix Laarmann, Jakob Rehof, and Christoph Stahl

TU Dortmund University, Dortmund, Germany
{andrej.dudenhefner,felix.laarmann,jakob.rehof,christoph.stahl}@cs.tu-dortmund.de

Introduction. In the 90s Dezani-Ciancaglini and Hindley presented *combinatory logic* [25] as a Hilbert-style calculus with *intersection types* [13]. This inspired the line of work [22, 23, 15, 14] by Rehof and Urzyczyn on composition synthesis based on intersection type inhabitation. Specifically, the idea of *combinatory logic synthesis* (CLS) is: Given a repository Γ of combinators typed by intersection types and an intersection type τ, construct combinatory terms M such that M is assigned the type τ wrt. Γ in Finite Combinatory Logic [22], written $\Gamma \vdash M : \tau$. There are several practical implementations of CLS in programming languages including C# [7], F# [16], Scala [2], Coq [3], and Python [4]. Most notably, CLS was applied effectively for synthesis of software product lines [18, 17, 6], simulation models [19], and motion planning programs [24]. More than a decade of empirical evaluation [3, Chapter 4] demonstrated the versatility of intersection types as component *specification language*. However, a notable weakness of intersection types as synthesis *query language* is the inability to express negative information [1, Chapter 5]. Motivated by the line of work by Castagna on programming with *set-theoretic types* [11], we propose a Boolean extension to the CLS query language, adding the connectives $\wedge$, $\vee$, and $\neg$. We give a stratified type system, consisting of a variant of finite combinatory logic [22, Section 4] and (partly) a monomorphic variant of the set-theoretic type system [11, Section 4]. We implement two distinct approaches [9, 8] for Boolean query evaluation, based on the most influential CLS frameworks (CLS-Scala [2] and CLS-Python [4]). Formally, we give inhabitation procedures for the presented stratified type system. Finally, we compare the two approaches and outline open questions.

Preliminaries. CLS-Scala and CLS-Python use intersection types with products and type constructors [3, Definition 3], given by the grammar $\sigma, \tau ::= \omega \mid c(\sigma) \mid \sigma \times \tau \mid \sigma \to \tau \mid \sigma \cap \tau$, where c ranges over unary, covariant, distributing constructors. Accordingly, intersection type subtyping $(\leq)$ is extended [3, Definition 5]. Combinatory terms $M, N ::= X \mid M\,N$, where X ranges over term variables, are assigned intersection types according to the rules of finite combinatory logic FCL$(\cap, \leq)$ [22, Figure 3], given below.

$$\frac{}{\Gamma, X : \tau \vdash X : \tau}\ (\textsc{Var}) \quad \frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau}\ (\cap\text{I}) \quad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M N : \tau}\ (\to\text{E}) \quad \frac{\Gamma \vdash M : \sigma \quad \sigma \leq \tau}{\Gamma \vdash M : \tau}\ (\leq)$$

We introduce the Boolean query language $\varphi, \psi ::= \sigma \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\varphi$, where intersection types act as propositions. For query evaluation we extend the above typing rules as follows.

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash_{\mathbb{B}} M : \tau}\ (\textsc{Embed}) \quad \frac{\Gamma \vdash_{\mathbb{B}} M : \varphi \quad \Gamma \vdash_{\mathbb{B}} M : \psi}{\Gamma \vdash_{\mathbb{B}} M : \varphi \wedge \psi}\ (\wedge\text{I}) \quad \frac{\Gamma \vdash_{\mathbb{B}} M : \varphi_i \quad i \in \{1, 2\}}{\Gamma \vdash_{\mathbb{B}} M : \varphi_1 \vee \varphi_2}\ (\vee\text{I}) \quad \frac{\Gamma \nvdash_{\mathbb{B}} M : \varphi}{\Gamma \vdash_{\mathbb{B}} M : \neg\varphi}\ (\neg\text{I})$$

The following example illustrates the interaction of layers $\vdash$ and $\vdash_{\mathbb{B}}$ in the above type system.

Example 1. Consider the repository $\Gamma = \big\{X : a \cap b \cap d, Y : d, F : (a \to b) \cap \big(d \to (a \cap c)\big)\big\}$ and the query $\varphi = a \wedge \neg(b \wedge c)$, where a, b, c, d abbreviate the types $a(\omega)$, $b(\omega)$, $c(\omega)$, $d(\omega)$. We have $\Gamma \vdash_{\mathbb{B}} M : \varphi$ iff $M \in \{X, F\,Y\}$. Since $\Gamma \vdash F\,X : a$, $\Gamma \vdash F\,X : b$, and $\Gamma \vdash F\,X : c$ we have that $\Gamma \nvdash_{\mathbb{B}} F\,X : \varphi$.

CLS-Scala approach. Given a repository Γ and an intersection type τ, the inhabitation procedure in CLS-Scala results in a regular tree grammar recognizing the set of combinatory terms $\{M \mid \Gamma \vdash M : \tau\}$. Such terms can be interpreted and evaluated as Scala programs. For minimal interference with existing projects and maximal code reuse, we conservatively extend CLS-Scala. We implement the connectives $\wedge$, $\vee$, and $\neg$ as tree grammar intersection, union, and complement relative to the set of inhabitants of the universal type ω. Both tree grammar intersection and union algorithms are well-known [12]. However, the relative complement requires a repository-aware completion (cf. [12, Example 1.1.6]) and complementation [12, Chapter 1.3]. For evaluation, we intersect the resulting grammar with the set of well-typed Scala terms.

Consider the repository Γ and the query $\varphi = a \wedge \neg(b \wedge c)$ from Example 1. The tree languages of inhabitants of the types a, b, and c are $\{X, F\,X, F\,Y\}$, $\{F\,(F\,Y), F\,X, F\,(F\,X), X\}$, and $\{F\,X, F\,Y\}$ respectively. By language intersection, the inhabitants of $b \wedge c$ are $\{F\,X\}$. Therefore, inhabitants of the query $\neg(b \wedge c)$ are in the relative complement of $\{F\,X\}$, which contains all combinatory terms built from the term variables X, Y, and F, except $F\,X$. Finally, the language intersection of the inhabitants of type a with those of the query $\neg(b \wedge c)$ is $\{X, F\,Y\}$.

CLS-Python approach. Rather than lifting the connectives to grammar operations, we extend the inhabitation procedure (INH) [22, Figure 4] to handle negative information. We observe that any Boolean query can be presented equivalently in minimal disjunctive normal form (DNF) [21, 20]. Therefore, it suffices to extend INH to work with conjunctive clauses, and combine the results. For this, we exclude inhabitants according to negative information (cf. INH, Line 5), and propagate negative information recursively (cf. INH, Line 8).

For the query φ from Example 1, a minimal DNF is $(a \wedge \neg b) \vee (a \wedge \neg c)$. The first clause $a \wedge \neg b$ is inhabited by terms of shape $F\,M$, for some term M of type d but not of type a. Propagating positive and negative information, the clause $d \wedge \neg a$ is constructed, which is inhabited by Y. The second clause $a \wedge \neg c$ is inhabited by X. In sum, the terms in $\{X, F\,Y\}$ inhabit the query φ.

Comparison. Both approaches require similar implementation effort of approx. 500 LOC.

The main advantage of the CLS-Scala approach is its modularity in two ways. First, as a conservative extension it does not interfere with existing projects. Second, it is not limited to Boolean connectives. Other tree language operations, such as a restriction wrt. term rewriting systems [5], can be addressed in this way. Additionally, computability follows from established results [12, 22]. The main disadvantage is an exponential blowup for intersection and complement operations, which negatively impacts scalability.

In comparison, the main advantage of the CLS-Python approach is a fine-grained control over information propagation in the inhabitation procedure, avoiding blowup in many cases. However, it is unclear how to address other operations.

A performance evaluation on the basis of a common scalable benchmark [10] for CLS has shown wildly different behavior of the presented approaches. Still, both approaches perform well in representative scenarios.

Open questions. The main open question is the exact relationship of the presented stratified type system and the family of set theoretic type systems [11]. This may provide insight on how to extend the specification language to include negative information. Complementarily, an investigation into syntactic subtyping [11, Section 3.1] in the full query language is essential. Further open questions regard the exact logic and model of the proposed type system, complexity of the inhabitation decision problem.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] Jan Bessai. (CL)S Scala. https://github.com/combinators/cls-scala. Accessed: 2023-03-01.

[3] Jan Bessai. *A type-theoretic framework for software component synthesis.* PhD thesis, Technical University of Dortmund, Germany, 2019.

[4] Jan Bessai, Constantin Chaumet, Anne Meyer, and Daniel Scholtyssek. (CL)S Python. https://cls-python.github.io/cls-python. Accessed: 2023-02-28.

[5] Jan Bessai, Łukasz Czajka, Felix Laarmann, and Jakob Rehof. Restricting Tree Grammars with Term Rewriting. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:19, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[6] Jan Bessai, Boris Düdder, George T. Heineman, and Jakob Rehof. Combinatory synthesis of classes using feature grammars. In Christiano Braga and Peter C. Ölveczky, editors, *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, volume 9539 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2015.

[7] Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens, and Jakob Rehof. Combinatory logic synthesizer. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, volume 8802 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2014.

[8] Jan Bessai, Andrej Dudenhefner, and Christoph Stahl. (CL)S Python with Boolean Queries. https://github.com/christofsteel/bcls-python. Accessed: 2023-03-03.

[9] Jan Bessai and Felix Laarmann. (CL)S Scala with Boolean Queries. https://github.com/FelixLaarmann/cls-scala. Accessed: 2023-03-03.

[10] Jan Bessai and Anna Vasileva. User support for the combinator logic synthesizer framework. *arXiv preprint arXiv:1811.10815*, 2018.

[11] Giuseppe Castagna. Programming with union, intersection, and negation types. *CoRR*, abs/2111.03354, 2021.

[12] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. *Tree automata techniques and applications.* 2008.

[13] Mariangiola Dezani-Ciancaglini and J. Roger Hindley. Intersection types for combinatory logic. *Theor. Comput. Sci.*, 100(2):303–324, 1992.

[14] Boris Düdder, Moritz Martens, and Jakob Rehof. Staged composition synthesis. In *Programming Languages and Systems: 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 23*, pages 67–86. Springer, 2014.

[15] Boris Düdder, Moritz Martens, Jakob Rehof, and Paweł Urzyczyn. Bounded combinatory logic. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 243–258. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[16] Andrej Dudenhefner. *Algorithmic aspects of type-based program synthesis.* PhD thesis, Technical University of Dortmund, Germany, 2019.

[17] George T. Heineman, Jan Bessai, Boris Düdder, and Jakob Rehof. A long and winding road towards modular synthesis. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th Interna-*

tional Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I, volume 9952 of *Lecture Notes in Computer Science*, pages 303–317, 2016.

[18] George T. Heineman, Armend Hoxha, Boris Düdder, and Jakob Rehof. Towards migrating object-oriented frameworks to enable synthesis of product line members. In Douglas C. Schmidt, editor, *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 56–60. ACM, 2015.

[19] Alexander Mages, Carina Mieth, Jens Hetzler, Fadil Kallat, Jakob Rehof, Christian Riest, and Tristan Schäfer. Automatic component-based synthesis of user-configured manufacturing simulation models. In *2022 Winter Simulation Conference (WSC)*, pages 1841–1852. IEEE, 2022.

[20] Edward J. McCluskey. Minimization of Boolean Functions. *The Bell System Technical Journal*, 35(6):1417–1444, 1956.

[21] Willard V. Quine. The problem of simplifying truth functions. *The American Mathematical Monthly*, 59(8):521–531, 1952.

[22] Jakob Rehof and Paweł Urzyczyn. Finite combinatory logic with intersection types. In C.-H. Luke Ong, editor, *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, volume 6690 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2011.

[23] Jakob Rehof and Paweł Urzyczyn. The complexity of inhabitation with explicit intersection. In Robert L. Constable and Alexandra Silva, editors, *Logic and Program Semantics - Essays Dedicated to Dexter Kozen on the Occasion of His 60th Birthday*, volume 7230 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2012.

[24] Tristan Schäfer, Jan Bessai, Constantin Chaumet, Jakob Rehof, and Christian Riest. Design space exploration for sampling-based motion planning programs with combinatory logic synthesis. In *Algorithmic Foundations of Robotics XV: Proceedings of the Fifteenth Workshop on the Algorithmic Foundations of Robotics*, pages 36–51. Springer, 2022.

[25] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische annalen*, 92(3-4):305–316, 1924.

A Formalization of Python's Execution Machinery

Ammar Karkour[1] and Giselle Reis[1]

Carnegie Mellon University, Qatar
akarkour@andrew.cmu.edu, giselle@cmu.edu

Motivation. Python's increasing popularity has led to its adoption from entry-level programmers to scientists and engineers [1, 3, 4]. Hence, a trustworthy Python execution machinery should be critical and valuable. However, the language lacks a comprehensive formal definition that could be used to provide provable guarantees and guide a verified implementation [7, 8]. Previous attempts to formalize Python's source code have left out several features and core parts. This is in part due to the sheer size, complexity, and constant evolution of the language. But also, Python's definition is written in natural language (e.g. English), which can be imprecise, open to interpretation, and inconsistent with the actual implementation [7, 8].

Contribution. We propose that, since Python's virtual machine executes bytecode, an alternative direction towards a verified Python implementation is to start from this lower, smaller and more stable level. Therefore, we present, to our knowledge, the first formalization of Python's bytecode and virtual machine. This is, of course, not free of challenges, as Python's bytecode specification is also written in natural language. When descriptions were not clear, we used `cpython` as Python's reference implementation to fully understand the semantics. Our formalization uses inference rules in the style of [5, 6] to define typing of objects and semantics, which includes bytecode execution and frame stack management. The proposed rules are shown to satisfy progress and preservation. In addition, our proposed framework can be extended with built-in types without breaking safety guarantees. The formalized rules were implemented in F$\star$, where properties can be proved automatically via dependent types or lemmas solved by an SMT solver. We call this implementation Py$\star$ [1]. From the F$\star$ implementation, we extracted a Python bytecode interpreter in OCaml. This verified Python execution machinery was compared to `cpython` for both consistency and performance.

Typing. Python is an object-oriented language in which all entities are objects of a certain class. However, unlike other object-oriented languages, there is no static class (or type) checking. Hence, one could say that all objects are of class `Object` statically, and their "real" class is only discovered at runtime. In that sense, Python is statically *unityped*, which means that "type checking" is now the responsibility of the execution machinery. Therefore, Python objects must have enough typing information so that the virtual machine is able to check types at runtime, and raise the appropriate errors when necessary. At the same time, this internal typing information should not impact how programmers see and operate with the objects. We achieve this by encapsulating the type information inside the top-level `Object` type.

Our typing system consists of 3 different layers `valTyp`, `cls`, and `pyObj`. At the innermost level is `valTyp`, indicating whether the class implements a built-in type (e.g. `int`, `string`, `list`, etc.) or is defined by the user (`USERDEF`). Below is a sample of the `valTyp` typing rules:

$$\frac{}{\texttt{USERDEF} : \texttt{valTyp}} \qquad \frac{i : \mathit{int}}{\texttt{INT}(i) : \texttt{valTyp}} \qquad \frac{s : \mathit{str}}{\texttt{STRING}(s) : \texttt{valTyp}} \qquad \frac{b : \mathit{bool}}{\texttt{BOOL}(b) : \texttt{valTyp}}$$

[1]The code for Py$\star$ can be found here https://github.com/ammarkarkour/PyStar/

A `valTyp` value is encapsulated in a `cls` record, which is the type of all objects in Python's source code. This record contains the name of the class, the process id of the object, a `valTyp` value, and two mappings of fields and methods.

When it comes to execution, `cpython` uses the same type for both source code and virtual machine objects. E.g., a bytecode instruction and an integer would both have type `PyObject`. However, the distinction between these two kinds of objects is crucial for our formalization, as it allows proving correctness of the virtual machine independently of the user's code. This distinction is made via the constructors of `pyObj`, which are: `PYTYP`(obj) for `cls` objects; `CODEOBJECT`(co) for bytecode; `FRAMEOBJECT`(f) for frames (i.e. a program state); `FUN`(f) for functions on built-in Python types (e.g. < or `__lt__`); and `ERR`(s) for errors. Below is a sample of `pyObj` typing rules (in the interest of space, we will not show the typing rule for `frameObj`):

$$\frac{obj : \texttt{cls}}{\texttt{PYTYP}(obj) : \texttt{pyObj}} \qquad \frac{msg : str}{\texttt{ERR}(msg) : \texttt{pyObj}} \qquad \frac{f : list\ \texttt{pyObj} \rightarrow \texttt{valTyp}}{\texttt{FUN}(f) : \texttt{pyObj}} \qquad \frac{f : \texttt{frameObj}}{\texttt{FRAMEOBJECT}(f) : \texttt{pyObj}}$$

Semantics. Python's execution machinery works on a stack of *frames*. A *frame* is a tuple $\langle \varphi, \Gamma, i, \Delta \rangle$ where φ is a name context, Γ contains the bytecode Π, i is the program counter, and Δ is the data stack. The semantic rules formalize how frames f are evaluated and how the frame stack K is managed. A frame stack in the evaluation state is written as $K \triangleright f$, and in the return state as $K \triangleleft \mathsf{ret}(v)$. Frame stack evaluation rules use the judgment $K \circ f \longmapsto K' \circ f$, where $\circ \in \{\triangleright, \triangleleft\}$. Frame evaluation rules use the judgment $f \xmapsto{\Gamma.\Pi[i]} f'$, where the arrow is labelled with the bytecode operation being executed. An example of each kind of rule is shown below:

$$\frac{\langle \varphi, \Gamma, i, \Delta \rangle \xmapsto{\Gamma.\Pi[i]} \langle \varphi_n, \Gamma_n, i_n, \Delta_n \rangle}{K \triangleright \langle \varphi, \Gamma, i, \Delta \rangle \longmapsto K \triangleright \langle \varphi_n, \Gamma_n, i_n, \Delta_n \rangle} \qquad \frac{}{\langle \varphi, \Gamma, i, v :: \Delta \rangle \xmapsto{\Gamma.\Pi[i]=\texttt{POP\_TOP}} \langle \varphi, \Gamma, i+1, \Delta \rangle}$$

Safety Proving safety (or soundness) of our typing system entails proving that well-typed terms do not reach a "stuck state", which is a state where no formal semantics rule is applicable [6]. This property is ensured by proving *progress* and *preservation* of our rules:

Thm 1 (Frame Stack Semantic Progress)**.** *A well-typed frame stack does not get stuck, that is, it is either in a final state or it can take a step according to the frame stack semantic rules.*

Thm 2 (Preservation)**.** *If an object $o : \tau$ evaluates to o', then $o' : \tau$.*

The proofs of the theorems follow the expected pattern. However, one must go through them to have at least a sanity check that all cases are covered.

Implementation. F⋆ is a general-purpose functional programming language with effects aimed at program verification [9]. One of the motivations for choosing F⋆ for our formalization was because the tool was successfully used to verify assembly instructions, which is a project close to ours [2]. Our verified implementation starts by embedding the defined types and objects in F⋆. Following that, we enforce the semantics rules' properties through the use of F⋆'s dependent types and lemmas. For example, this is how the rule for `POP_TOP` is implemented:

```
val pop_top: (l: list pyObj {Cons? l}) -> Tot (l2: list pyObj {l2 == tail l})
let pop_top datastack = List.Tot.Base.tail datastack
```

Following that, we use F⋆'s tools to extract a verified Python bytecode interpreter in OCaml, which was tested against hand-crafted test cases and a subset of `cpython`'s test kit. We are actively working on covering the whole of `cpython`'s test suite.

References

[1] Martin Desharnais and Stefan Brunthaler. Towards Efficient and Verified Virtual Machines for Dynamic Languages. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 61–75. Association for Computing Machinery, 2021.

[2] Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A Verified, Efficient Embedding of a Verifiable Assembly Language. *Proc. ACM Program. Lang.*, 3(POPL), 2019.

[3] Samuel Groß. JITSploitation I: A JIT Bug. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html, 2020. Accessed: 2021-12-10.

[4] Samuel Groß. JITSploitation III: Subverting Control Flow. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-three.html, 2020. Accessed: 2021-12-10.

[5] Robert Harper. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press, 2016.

[6] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[7] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: The Full Monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, page 217–232. Association for Computing Machinery, 2013.

[8] Python. Dis - disassembler for python bytecode. https://docs.python.org/3.8/library/dis.html#module-dis, 2022. Accessed: 2022-10-11.

[9] Nikhil Swamy, Catalin Hritcu, 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.

Games and Strategies using Coinductive Types

Peio Borthelle[1], Tom Hirschowitz[2], Guilhem Jaber[3], and Yannick Zakowski[4]

[1] Université Savoie Mont Blanc, LAMA, Chambéry, France
[2] CNRS, LAMA, Chambéry, France
[3] Nantes Université, LS2N, Nantes, France
[4] Inria, LIP, Lyon, France

Introduction Bisimulations and game techniques for higher-order languages have proved to be powerful tools for reasoning about program equivalence and building models that scale to advanced features such as side effects or existential types. Yet, their usage in mechanized proofs is rare. In this work in progress, we argue that this observation is, in part, the consequence of several mismatches between the traditional presentation of games in set theory and idiomatic constructions from type theory. We hence present a formulation of games and strategies more amenable to manipulation in proof assistants.

The framework we propose is structured around a coinductive representation of labelled transition systems (LTS), inspired by *interaction structures* [HH06], by the Coq library of *interaction trees* [XZH+20], and building upon the work of Levy and Staton [LS14]. Our main contribution is to provide a unified account of *operational game semantics* (OGS [Lai07, LL07]), an LTS-based game model for which we prove the correctness of the generated bisimulation with respect to contextual equivalence[1]. The construction, and the proof, are parametrized by a rather loose notion of evaluator assumed to satisfy a succinct axiomatization. In this talk, we will focus on (1) introducing the standard approach of operational game semantics succinctly, before (2) giving a more detailed account of the peculiarities and advantages of our representation of games and strategies.

Operational Game Semantics The behavior of a program can be represented as the set of its interactions with any execution environment. These sets of interactions can be generated intensionally by an LTS, where the labels encode information exchanged. For higher-order languages, this interaction may typically be the application of the term at hand to an arbitrary value v. One might be tempted to describe it as the transition $\lambda x.e \xrightarrow{\mathbf{app}(v)} e[x \mapsto v]$ but embarking higher order values in labels leads to challenging notions of bisimilarity. Following a technique used in pointer-games [HO00], operational game semantics provides a way to keep the traces first-order: instead of full-blown terms, only an *abstracted* or inert version is exchanged, with fresh *channel names* in place of subterms we wish to hide. The LTS of a term is constructed by evaluating it to a normal form, say $E[x\,v]$ in call-by-value, and issuing a label corresponding to the shape of this normal form. Here the label $\mathbf{app}(x)$ is issued and will bind two fresh channels, one for the abstracted argument v provided to x and one for the abstracted continuation E. This transition leads the LTS to a passive state where the environment (Opponent) is able to resume computation by choosing an available channel.

Labels are now semantically simpler, but they bind and reference channel names. To tackle this, we resort to a static scoping discipline and use dependent-types. Labels are indexed by channel scoping information, containing types such as $s \to t$ for functions and $\neg s$ for continua-

[1]At the time of writing the mechanized version of the proof is not complete.

tions. The function **next** gives the new scope after a label has been issued (slightly simplified):

$$\begin{array}{ll} \textbf{label} : \text{scope} \to \text{Type} & \textbf{next}_\Gamma : \text{label } \Gamma \to \text{scope} \\ \textbf{label } \Gamma \coloneqq \textbf{app}_{s,t}\ (s \to t \in \Gamma) \mid \textbf{ret}_s\ (\neg s \in \Gamma) & \textbf{next}_\Gamma\ (\text{app}_{s,t}\ i) \coloneqq s, \neg t, \Gamma \\ & \textbf{next}_\Gamma\ (\text{ret}_s\ i) \coloneqq s, \Gamma \end{array}$$

The astute reader will have recognized that these label and scope transition rules already form an LTS! We dub it the *game specification*. The OGS LTS proper is indexed over this specification LTS. As our languages of interest have general recursion, we allow usage of the delay monad $\mathcal{D}$ [Cap05]. We give the types of the configurations and active/passive transition functions:

$$\begin{array}{ll} \textbf{conf-act} : \text{scope} \to \text{Type} & \textbf{trans-act} : \text{conf-act } \Gamma \to \mathcal{D}((m : \text{label } \Gamma) \times \text{conf-pas } (\text{next}_\Gamma\ m)) \\ \textbf{conf-pas} : \text{scope} \to \text{Type} & \textbf{trans-pas} : \text{conf-pas } \Gamma \to (m : \text{label } \Gamma) \to \text{conf-act } (\text{next}_\Gamma\ m) \end{array}$$

From Polynomial Functors to Two-Player Games OGS is a symmetric game but in general, Proponent-chosen and Opponent-chosen labels might be different. Thus our two-player game specifications consist of two matching *half-game* descriptions. Descriptions are parametrized by a set of *states* for each player, each side giving for each state the set of allowed moves, and for each move the next state. Each half-game gives rise to two functors on families which we call the *active* and *passive* interpretation.

$$\begin{array}{l} \text{record } \textbf{half-game}\ (I\ J : \text{Type}) \coloneqq \{\, \textbf{move} : I \to \text{Type} \,;\ \textbf{trans} : \forall i, \text{move } i \to J \,\} \\ \text{record } \textbf{game}\ (I\ J : \text{Type}) \coloneqq \{\, \textbf{ply} : \text{half-game } I\ J \,;\ \textbf{opp} : \text{half-game } J\ I \,\} \\ \textbf{active}\ (H : \text{half-game } I\ J)\ (X : J \to \text{Type})\ i \coloneqq (m : H.\text{move } i) \times X(H.\text{trans } m) \\ \textbf{passive}\ (H : \text{half-game } I\ J)\ (X : J \to \text{Type})\ i \coloneqq (m : H.\text{move } i) \to X(H.\text{trans } m) \end{array}$$

An indexed polynomial endofunctor [AGH+15] can be constructed by composing the active resp. passive interpretation of Proponent resp. Opponent half-games. We can then build strategies by taking an infinite tree construction on this endofunctor. As we wish to handle looping in strategies, following the lead of interaction trees, our construction of choice is a free complete Elgot monad [GMR16] which we give here in two mutually coinductive definitions. The three cases in active strategies correspond respectively to leaves, silent steps similar to the "later" node of the delay monad, and playing a move. Passive strategies correspond to waiting for an Opponent move.

$$\begin{array}{l} \textbf{strat}^+ (G : \text{game } I\ J)\ X\ i \coloneqq \textbf{ret}\ (X\ i) \mid \textbf{tau}\ (\text{strat}^+\ G\ X\ i) \mid \textbf{vis}\ (\text{active } (G.\text{ply})\ (\text{strat}^-\ G\ X)\ i) \\ \textbf{strat}^-\ (G : \text{game } I\ J)\ X\ j \coloneqq \text{passive } (G.\text{opp})\ (\text{strat}^+\ G\ X)\ j \end{array}$$

Several dualities are at play. First, we can dualize a game by swapping the two components, hence reversing the players' roles. This dualization is definitionally involutive, an improvement over [XZH+20, HH06] where question-answer swapping is hard to make sense of. Second, on half-games there is a functor-functor interaction law [KRU20] between the passive and active interpretations which we dub *synchronization*. Intuitively it makes a sender and a receiver interact and progress. These two constructions together give rise to several more or less general composition operators between strategies and counter-strategies, of which we give a simple one:

$$\begin{array}{l} \textbf{sync} : \Sigma_i(\text{active } H\ X\ i \times \text{passive } H\ Y\ i) \to \Sigma_j(Xj \times Yj) \\ \textbf{compo} : \Sigma_i(\text{strat}^+\ G\ X\ i \times \text{strat}^-\ G^\perp\ Y\ i) \to \mathcal{D}(\Sigma_i X i + \Sigma_j Y j) \end{array}$$

Moreover, like polynomial functors, half-games and games are closed under a number of combinators, some studied in [LS14] and strikingly similar to linear logic connectives which we have to investigate further.

References

[AGH$^+$15] Thorsten Altenkirch, Neil Ghani, Peter G. Hancock, Conor McBride, and Peter Morris. Indexed containers. *J. Funct. Program.*, 25, 2015.

[Cap05] Venanzio Capretta. General recursion via coinductive types. *Log. Methods Comput. Sci.*, 1(2), 2005.

[GMR16] Sergey Goncharov, Stefan Milius, and Christoph Rauch. Complete elgot monads and coalgebraic resumptions. *Electronic Notes in Theoretical Computer Science*, 325:147–168, 2016. The Thirty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXII).

[HH06] Peter Hancock and Pierre Hyvernat. Programming interfaces and basic topology. *Ann. Pure Appl. Log.*, 137(1-3):189–239, 2006.

[HO00] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: i, ii, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[KRU20] Shin-ya Katsumata, Exequiel Rivas, and Tarmo Uustalu. Interaction laws of monads and comonads. In *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 604–618. ACM, 2020.

[Lai07] James Laird. A fully abstract trace semantics for general references. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 667–679. Springer, 2007.

[LL07] Søren B. Lassen and Paul Blain Levy. Typed normal form bisimulation. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2007.

[LS14] Paul Blain Levy and Sam Staton. Transition systems over games. In *Proceeding of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 64:1–64:10. ACM, 2014.

[XZH$^+$20] 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. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020.

11

Session 13: Formalizing mathematics using type theory

Formalized Non-wellfounded Syntax through 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

Motivation. Lambda-calculi are term structures involving variable binding. Untyped lambda-terms, in particular, have very often been extended to potentially *non-wellfounded* lambda-terms; these still consist of variables, lambda-abstractions and applications only, but the construction process can go on forever. Such construction processes (e. g., for Böhm trees) can nevertheless be described through functional programming, and the host programming language then serves as a meta-language for the description of those infinitary lambda-terms.

We do not only want to be able to program with such structures but aim to develop a fully formalized theory about them. For the formalization, we have chosen the Coq system [16].

Coq features a built-in mechanism for specifying coinductive types and for defining functions by corecursion. However, definitions by corecursion in Coq face numerous issues with guardedness, in particular when the coinductive type makes also use of a parameterized inductive type whose parameter is built with the coinductive type. (This is a so-called mixed inductive-coinductive definition, see a recent PhD thesis on the topic [5]; workarounds especially for *proofs* by coinduction exist through Mendler's style [13], but in the present work, we only rely on the universal property of a final coalgebra.)

Our formalization of non-wellfounded syntax is developed within formalized category theory in the UniMath library of univalent mathematics [17] on top of Coq. We are using UniMath for its large library of category theory; although our development is informed by univalent foundations, we still mimic ordinary category-theoretic constructions, such as the construction of final coalgebras as ω-limits.

An application scenario. There are many application scenarios for potentially non-wellfounded syntax with variable binding. We focus here on one [14, Section 3.2] where the second author has been involved, and which we plan to study with our formalization: the idea is to represent the entire search space for inhabitants in simply-typed lambda-calculus by a potentially non-wellfounded term of a suitable calculus. This calculus is informally given by the following grammar:

$$\begin{array}{lll} \text{(terms)} & N & ::=_{co} \quad \lambda x^A.N \,|\, E_1 + \cdots + E_n \\ \text{(elimination alternatives)} & E & ::=_{co} \quad x\langle N_1, \ldots, N_k\rangle \end{array}$$

where both $n, k \geq 0$ are arbitrary. The elements of the syntactic category of terms are also called *forests*. The index *co* means that the grammar is read coinductively. The typing rule for the sums of E's is just that all summands have to have the same type, and this is the type of the sum—it represents alternatives. The other rules are inherited from simply-typed λ-calculus. Let 0 be a base type. The closed forest Nat of type $(0 \to 0) \to 0 \to 0$ is given by $\mathsf{Nat} := \lambda f^{0\to 0}\lambda x^0.N$, with N of type 0 coinductively given by $N = x\langle\rangle + f\langle N\rangle$. This is a representation of all Church numerals, including infinity. The corecursive equation for N is appreciated on the level of the host programming language. The cited paper notably associates with every simple type A a forest that represents the entire search space for inhabitants in long normal form of A. Typing plays a major role here, so a formalization has to take into account the typing. The types of the object language (for which we give a deep embedding into Coq) will henceforth be called sorts, whence we speak about multi-sorted languages with (sorted) variable binding. But already the untyped forests above profit from multi-sortedness: we need to use three sorts corresponding to the three syntactic entities of variables, terms, and elimination alternatives.

As a benefit of the deep embedding, we obtain a generic construction for all descriptions of multi-sorted coinductive term calculi with binding (and thus have no need for meta-programming to reason about all of them).

The actual construction. The following steps are done uniformly for all languages we can express in our setting. (i) We describe simply-typed syntax with variable binding (of finitely many sorted variables in each constructor argument) as a multi-sorted binding signature. (ii) We then construct a signature functor H (deviating from [4] for technical reasons), with proofs of ω-continuity of H for the coinductive syntax, and a "lax lineator" between actions expressing pointed tensorial strength of H. (iii) We construct the coinductive syntax as inverse of a final coalgebra (using ω-continuity). (iv) We construct a generalized substitution operation (a *generalized heterogeneous substitution system*—a new abstraction that works both for inductive and coinductive syntax). (v) We construct a Σ-monoid (a notion relative to monoidal categories—this is generic for all generalized heterogeneous substitution systems). (vi) Finally we interpret the obtained monoid as monad (hence as monadic substitution) by instantiating the monoidal category to the endofunctors. These steps are formalized in UniMath [17].[1]

Our construction builds on previous work; the work we use most directly is the following: [12] for the construction of a well-behaved substitution system for wellfounded and non-wellfounded syntax; [4] for the chain from multi-sorted binding signatures to certified monadic substitution on the wellfounded syntax, formalized in UniMath; [9] for the abstraction level of a monoidal category (and hence the construction of a substitution monoid rather than a substitution monad)—multi-sorted syntax is not considered there; and [8] and [10] for the actegorical notion of strength suitable to incorporate pointedness (that plays the role of the insertion of variables into terms). We could have made use of the Coq code [11] for skewed monoidal categories and Σ-monoids, but we redeveloped notions in UniMath that profit more from the mechanism of displayed categories, functors, etc., for the construction and analysis of layered structures available in UniMath since [3].

Final comments. The work we are describing resides on different levels of abstraction. From the point of view of formalization, the taken levels seem to be a good compromise, as evidenced by the fact that the full formalization already exists in UniMath. However, in several dimensions, there could be more generality: Our coinductive types are rather coinductive families of sets and not of higher homotopy level (such as the construction of M-types in [1], which cannot represent variable binding). Pseudo-algebras for pseudo-monads are the abstract concept behind monoidal categories and actegories, as explained in depth in [7]. Furthermore, the variable binding that can be captured by multi-sorted binding signatures is rather concrete, while an abstract concept is based (again) on pseudo-monads and pseudo-distributive laws [15]. For a formalization in Coq (via the UniMath library), it would have to be seen if those notions could be suitably adapted from the strict two-categorical setting to bicategories for which an extensive library has been available in UniMath since [2].

[1]The development that is specific for the present abstract has been mostly done within the scope of the following pull request: `https://github.com/UniMath/UniMath/pull/1633`.

References

[1] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in homotopy type theory. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, volume 38 of *LIPIcs*, pages 17–30. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[2] Benedikt Ahrens, Dan Frumin, Marco Maggesi, Niccolò Veltri, and Niels van der Weide. Bicategories in univalent foundations. *Math. Struct. Comput. Sci.*, 31(10):1232–1269, 2021.

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

[4] 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.

[5] Henning Basold. *Mixed Inductive-Coinductive Reasoning—Types, Programs and Logic*. PhD thesis, Radboud University, Nijmegen, The Netherlands, 2018.

[6] Peio Borthelle, Tom Hirschowitz, and Ambroise Lafont. A cellular Howe theorem. 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 273–286. ACM, 2020.

[7] Matteo Capucci and Bruno Gavranović. Actegories for the working amthematician, 2022. https://arxiv.org/abs/2203.16351.

[8] Marcelo P. Fiore. Second-order and dependently-sorted abstract syntax. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 57–68. IEEE Computer Society, 2008.

[9] Marcelo P. Fiore, Gordon D. Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 193–202. IEEE Computer Society, 1999.

[10] Chung-Kil Hur. *Categorical equational systems : algebraic models and equational reasoning*. PhD thesis, University of Cambridge, UK, 2010.

[11] Ambroise Lafont. Initial sigma-monoids for skew monoidal categories in UniMath, 2022. This consists of Coq code that was initially a supplement to [6].

[12] Ralph Matthes and Tarmo Uustalu. Substitution in non-wellfounded syntax with variable binding. *Theoretical Computer Science*, 327(1-2):155–174, 2004.

[13] Keiko Nakata and Tarmo Uustalu. Resumptions, weak bisimilarity and big-step semantics for while with interactive I/O: An exercise in mixed induction-coinduction. In Luca Aceto and Pawel Sobocinski, editors, *SOS*, volume 32 of *EPTCS*, pages 57–75, 2010.

[14] José Espírito Santo, Ralph Matthes, and Luís Pinto. A coinductive approach to proof search through typed lambda-calculi. *Ann. Pure Appl. Log.*, 172(10):103026, 2021.

[15] Miki Tanaka and John Power. A unified category-theoretic formulation of typed binding signatures. In *Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, MERLIN '05, pages 13–24, New York, NY, USA, 2005. ACM.

[16] The Coq Development Team. The Coq proof assistant, version 8.16, September 2022.

[17] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath — a computer-checked library of univalent mathematics. Available at http://unimath.github.io/UniMath/ , 2023.

Categorical Logic in Lean

Jacob Neumann[1]

University of Nottingham
jacob.neumann@nottingham.ac.uk

Categorical Logic is a branch of mathematical logic which uses the concepts and tools of category theory to investigate logical systems and deductive calculi, following in the example of Lawvere's pioneering work on functorial semantics for algebraic theories[Law63]. In this talk, we'll provide a progress report on a formalization of categorical logic in the Lean proof assistant [dMKA+15][1]. Lean is an interactive theorem prover and dependently-typed functional programming language, based on the Calculus of Inductive Constructions. Proofs in Lean are done using *proof tactics*, making use of Lean's powerful and flexible `tactic` monad. In Lean, we can define new tactics – allowing for abstraction and reuse of common reasoning patterns–, and also make use of various tactic combinators to automate and simplify proofs.

As a simple proof-of-concept for categorical logic in Lean, we'll discuss the formalization of the *syntactic category* construction for the *positive propositional calculus (PPC)*, which is the following fragment of intuitionistic propositional logic: formulas are given by the grammar

$$\varphi, \psi \quad ::= \quad \mathsf{p} \mid \top \mid \varphi \wedge \psi \mid \varphi \to \psi$$

with the usual natural deduction rules for these connectives. By quotienting the set of formulas by the *inter-derivability* relation $\dashv\vdash$, we obtain the *syntactic poset* or *Lindenbaum-Tarski algebra* [Tar83] of the PPC. Viewing this poset as a category, we obtain the syntactic category of PPC. With the right Lean tactics, we're able to prove in just a few lines that this syntactic category forms a *cartesian closed category* (a key step in the proof of the completeness of the PPC with respect to Kripke semantics), with this extra categorical structure arising from the deductive rules of PPC. The full proof can be seen in Figure 1. Take for instance this line,

```
pr2 := by LiftT `[ apply And.and_elimr ],
```

which constructs the 'pairing' operation in a CCC (combining morphisms $f : Z \to X$ and $g : Z \to Y$ into $\langle f, g\rangle : Z \to X \times Y$) by *lifting* the PPC deduction rule of $\wedge$-introduction (if $\Phi \vdash \varphi$ and $\Phi \vdash \psi$, then $\Phi \vdash \varphi \wedge \psi$). The `LiftT` tactic defined as part of this project – whose operation we'll seek to describe – allows us to perform these kinds of 'liftings' of deduction rules onto constructions in the syntactic category. Time permitting, we will also discuss extensions to this basic PPC framework – such as the addition of modal operators $\Box$ and $\diamond$ to the logic, which correspond to (co)monads on the syntactic category – as well as the role this construction plays in soundness and completeness proofs.

The (work-in-progresss) documentation for this formalization project can be found at lean-catLogic.github.io.

[1]This formalization is done in Lean 3, and partially uses the accompanying mathematical library [mC20].

```

instance syn_FP_cat {Form : Type} [And : has_and Form] : FP_cat (Form _eq) :=
{
  unit := syn_obj And.top,
  term := by LiftT `[ apply And.truth ],
  unit_η := λ X f, by apply thin_cat.K,
  prod := and_eq,
  pr1 := by LiftT `[ apply And.and_eliml ],
  pr2 := by LiftT `[ apply And.and_elimr ],
  pair := by LiftT `[ apply And.and_intro ],
  prod_β1 := λ X Y Z f g, by apply thin_cat.K,
  prod_β2 := λ X Y Z f g, by apply thin_cat.K,
  prod_η :=  λ X Y, by apply thin_cat.K
}
instance syn_CC_cat {Form : Type} [Impl : has_impl Form] : CC_cat (Form _eq) :=
{
  exp := impl_eq,
  eval := by LiftT `[ apply cart_x.modus_ponens ],
  curry := by LiftT `[ apply cart_x.impl_ε],
  curry_β := λ {X Y Z} u, by apply thin_cat.K,
  curry_η := λ {X Y Z} v, by apply thin_cat.K,
```

Figure 1: For any deductive calculus with truth, conjunction, and implication (satisfying the usual rules), its syntactic category is a CCC. As discussed, the uses of `LiftT` are instances where deduction rules of the PPC are lifted to constructions on the syntactic category. The lines invoking `thin_cat.K` are appeals to the fact that the syntactic category is a poset in order to prove that certain diagrams commute.

References

[dMKA+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 378–388. Springer, 2015.

[Law63] F William Lawvere. Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences*, 50(5):869–872, 1963.

[mC20] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[Tar83] Alfred Tarski. *Logic, semantics, metamathematics: papers from 1923 to 1938.* Hackett Publishing, 1983.

Rezk Completion of Bicategories

Niels van der Weide[1] and Kobe Wullaert[2]

[1] Radboud University, The Netherlands
[2] Delft University of Technology, The Netherlands

Various kinds of structured categories are used to study the semantics of various flavors of type theory. For example, cartesian closed categories and symmetric monoidal closed categories are used to study the semantics of simple and linear type theory, respectively [3, 5]. Such categories represent models of the theory in study, whereas the initial such model represents the syntax.

If we would like to replicate this idea in univalent foundations, then we stumble on a difficulty. Categorically, one would describe the syntax as the initial model in the category of models. Since the correct notion of category in univalent foundations is that of a univalent category, this reflects onto the models as well. For the simply typed lambda calculus, this means that one needs to construct the initial univalent cartesian closed category. However, if one uses the usual presentation of the syntax (of, for example, the simply typed lambda calculus), then the acquired category is not guaranteed to be univalent. As such, one needs to do some extra work to acquire the desired initial model.

The solution for our problem, lies in what is known as the *Rezk completion*. In [2], it is shown how every category is weakly equivalent to a univalent category. However, in that paper, preservation of categorical structure under Rezk completion is not considered. More concretely, if a category has a cartesian closed or a symmetric monoidal structure, can we say the same about its Rezk completion?

The goal of this abstract is to study the Rezk completion of categories with some additional structure. More specifically, our goal is to show how, for some notion of structure, the Rezk completion of a category preserves the structure.

1 Displayed Universal Arrows

To generalize the Rezk completion to structured categories, there are several aspects that we need to consider. Among these aspects are the notion of structure in consideration and the desired universal property. For that purpose, we recall the universal property of the Rezk completion. Let us denote the Rezk completion of $\mathcal{C}$ by $\mathsf{RC}(\mathcal{C})$. In [2] it is shown how this univalent category $\mathsf{RC}(\mathcal{C})$ satisfies the following universal property: every functor $F : \mathcal{C} \to \mathcal{D}$ to a univalent category $\mathcal{D}$ factorizes uniquely through $\mathsf{RC}(\mathcal{C})$.

We can formulate this universal property in the language of bicategories. Given a type-theoretic universe $\mathcal{U}$, we write $\mathsf{Cat}_{\mathcal{U}}$ and $\mathsf{UnivCat}_{\mathcal{U}}$ for the bicategories of categories and of univalent categories in universe $\mathcal{U}$ respectively. Now suppose that RC preserves the universe level. If we assume that $\mathcal{U}$ is closed under a suitable class of higher inductive types, then we can construct the desired Rezk completion as a higher inductive type [6]. Then the universal property of the Rezk completion says that the inclusion $\mathsf{UnivCat}_{\mathcal{U}} \to \mathsf{Cat}_{\mathcal{U}}$ has a left biadjoint. By formulating this biadjunction in the language of universal arrows, we obtain the universal property mentioned before.

The next question is what we mean by a structured category, and to answer that, we use displayed bicategories [1]. Recall that a displayed bicategory over $\mathcal{B}$ corresponds to structure and properties to be added to the objects, 1-cells, and 2-cells of $\mathcal{B}$. As such, we represent a notion of structured categories by a displayed bicategory $\mathcal{D}$ over $\mathsf{Cat}_{\mathcal{U}}$. Note that every such displayed bicategory gives rise to a displayed bicategory $\mathcal{D}_{\mathsf{univ}}$ over $\mathsf{UnivCat}_{\mathcal{U}}$. The total bicategories $\int \mathcal{D}$ and $\int \mathcal{D}_{\mathsf{univ}}$ are the bicategories of structured categories and structured univalent categories respectively. Now we define when a notion of structured category admits a Rezk completion.

Definition 1. We say that $\mathcal{D}$ **admits a Rezk completion** if the inclusion pseudofunctor from $\int \mathcal{D}_{\mathsf{univ}}$ into $\int \mathcal{D}$ has a left biadjoint.

Note that constructing left biadjoints can be a rather demanding task due to the large amount of data and properties involved. A convenient tool for constructing biadjoints is *universal arrows* [4]. Since we work in a displayed setting, the notion of *displayed universal arrow* is more suitable for our setting.

Definition 2. Suppose that we have bicategories $\mathcal{B}_1$ and $\mathcal{B}_2$. Let $R : \mathcal{B}_1 \to \mathcal{B}_2$ be a pseudofunctor, and suppose that we have a left universal arrow L, whose unit we denote by $\eta : \mathsf{Id} \Rightarrow L \cdot R$. Let $\mathcal{D}_i$ be a displayed bicategory over $\mathcal{B}_i$ $(i = 1, 2)$ and $R^{\mathcal{D}}$ a displayed pseudo-functor $\mathcal{D}_1 \to \mathcal{D}_2$ over R. A displayed universal arrow of $R^{\mathcal{D}}$ over $L \dashv R$ consists of the following data:

1. A function $L^{\mathcal{D}} : \prod_{x:\mathcal{B}_2} (\mathcal{D}_2)_x \to (\mathcal{D}_1)_{Lx}$
2. For any $x : \mathcal{B}_2$ and $\bar{x} : (\mathcal{D}_2)_x$, a displayed morphism $\bar{x} \to_{\eta x} R^{\mathcal{D}}(L^{\mathcal{D}}(\bar{x}))$.

such that we have a certain displayed adjoint equivalence between displayed hom-categories.

Every displayed universal arrow gives rise to a universal arrow on the total bicategory, which in turn gives rise to a left biadjoint. As such, to show that $\mathcal{D}$ admits a Rezk completion, it is sufficient to construct a displayed universal arrow for the inclusion.

2 Rezk completions of structured categories

The aim of this work in progress is to show that a wide class of structured categories admits a Rezk completion. Our approach is modeled after the construction of the monoidal Rezk completion [7]. The idea is that the same steps as in [7] can be used for other structures. For that reason, we recall the steps taken in that proof.

Let $(\mathcal{C}, \otimes, I)$ be a monoidal category and denote by $\eta_{\mathcal{C}} : \mathcal{C} \to \mathsf{RC}(\mathcal{C})$ the weak equivalence given by the Rezk completion. Observe that the monoidal Rezk completion lifts the monoidal structure on $\mathcal{C}$ to $\mathsf{RC}(\mathcal{C})$ in such a way that $\eta_{\mathcal{C}}$ preserves the monoidal structure strongly. In addition, for every univalent monoidal category $\mathcal{D}$, the lifting $\mathsf{lift}_{\eta_{\mathcal{C}}}(F) : \mathsf{RC}(\mathcal{C}) \to \mathcal{D}$, of a monoidal functor $F : \mathcal{C} \to \mathcal{D}$ has a monoidal structure. This lifting is unique with respect to the equation $F \cong_{\text{monoidal}} \eta_{\mathcal{C}} \cdot \mathsf{lift}_{\eta_{\mathcal{C}}}(F)$, which is an isomorphism in the monoidal functor category $[\mathcal{C}, \mathcal{D}]_{monoidal}$. As such, we obtain an isomorphism of monoidal categories

$$\eta_{\mathcal{C}} \cdot - : [\mathsf{RC}(\mathcal{C}), \mathcal{D}]_{\text{monoidal}} \to [\mathcal{C}, \mathcal{D}]_{\text{monoidal}}.$$

Each of these three steps is (equivalently) proven using the fact that precomposing with $\eta_{\mathcal{C}}$ induces an adjoint equivalence of hom-categories.

Note that we can express the above proof using the language of Section 1. For this reason, we conjecture that the notions in Section 1 provide a setting in which we can generalize the Rezk completion to structured categories.

References

[1] Benedikt Ahrens, Dan Frumin, Marco Maggesi, Niccolò Veltri, and Niels van der Weide. Bicategories in univalent foundations. *Math. Struct. Comput. Sci.*, 31(10):1232–1269, 2021.

[2] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Math. Struct. Comput. Sci.*, 25(5):1010–1039, 2015.

[3] P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models (extended abstract). In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, volume 933 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1994.

[4] Thomas M Fiore. *Pseudo limits, biadjoints, and pseudo algebras: categorical foundations of conformal field theory.* American Mathematical Society, 2006.

[5] Paul-André Mellies. Categorical semantics of linear logic. *Panoramas et syntheses*, 27:15–215, 2009.

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

[7] Kobe Wullaert, Ralph Matthes, and Benedikt Ahrens. Univalent monoidal categories. *arXiv preprint arXiv:2212.03146*, 2022.

Towards quotient inductive-inductive-recursive types

Ambrus Kaposi

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

Induction-recursion [8] means that an inductive type is defined mutually with a recursive function from that type into a different type (left hand side).

$$\begin{array}{ll} \mathsf{data\,A : Set} & \qquad \mathsf{data\,A : Set} \\ \mathsf{f : A \to B} & \qquad \mathsf{f : A \to A} \end{array}$$

If B is small, this type can be reduced to an inductive type indexed over B [10] where the index of a constructor is given by the result of f on it. The above reduction does not work however when the return type of f is A itself (right hand side). A concrete example for such a type is the intrinsic syntax of type theory where instantiation of substitution is defined recursively [7]. The inductive parts of the definition are the separate sorts of contexts, parallel substitutions, types and terms, the recursive part is instantiation of substitution. This is a function which takes a term in context Γ and a substitution from Δ to Γ and returns a term in context Δ. We don't have semantics for these inductive-recursive types, but we are not surprised that Agda supports (some of) them. We don't know how to circumscribe the supported subset of such definitions other than saying that Agda's pattern matching mechanism and termination checker should accept them.

The syntax of type theory was defined using inductive-inductive types [5] before the notion induction-induction [9] was coined. Analogously, quotient inductive-inductive types (QIITs) were used to define the syntax of type theory [2] before the well-behaved QIITs were circumscribed [11]. Is there a way to define what a QII*R*T is?

In this talk we will give a completely precise definition of a *particular* QIIRT in an ordinary type theoretic metatheory and not relying on pattern matching or termination checking. This QIIRT is the intrinsic quotiented syntax of simple type theory. By intrinsic we mean that only well-formed, well-scoped, well-typed terms are part of the syntax, we don't even mention preterms or typing relations. By quotiented we mean that α-β-η-convertible terms are equal (up to Agda's equality type), in particular we don't even mention congruence rules (they are provable using the eliminator of the equality type).

Extrinsic syntax has the advantage over intrinsic syntax that instantiation is defined recursively, and thus equations such as $(t \cdot u)[\sigma] = (t[\sigma]) \cdot (u[\sigma])$ hold by definition ($- \cdot -$ denotes function application). The syntax we present below has the same feature, so this ceases to be an advantage anymore. In Agda (and even more in other proof assistants), extrinsic syntax still holds the advantage of avoiding transport hell, c.f. impressive extrinsic formalisations [1, 13].

We define models of simple type theory as simply typed categories with families (sCwFs [4]) with a base type and function space. We denote the category of contexts and substitutions by Con, Sub, $- \circ -$, id, instantiation of substitution by $-[-] : \mathsf{Tm}\,\Gamma\,A \to \mathsf{Sub}\,\Delta\,\Gamma \to \mathsf{Tm}\,\Delta\,A$, context extension by $- \rhd - : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Con}$, projections by $\mathsf{p} : \mathsf{Sub}\,(\Gamma \rhd A)\,\Gamma$ and $\mathsf{q} : \mathsf{Tm}\,(\Gamma \rhd A)\,A$, function space by $- \Rightarrow -$, lambda and application by the natural isomorphism $\mathsf{lam} : \mathsf{Tm}\,(\Gamma \rhd A)\,B \cong \mathsf{Tm}\,\Gamma\,(A \Rightarrow B) : -[\mathsf{p}] \cdot \mathsf{q}$. The *weak syntax* is the initial such model which can be defined as a QIIT. Its iteration principle says that there is a strict homomorphism from the weak syntax to any model ("strict" means that preservation of operations is definitional). Its induction principle says that there is a strict section of any displayed model over the weak syntax. Cubical Agda [15] supports the weak syntax, or it is possible to postulate the weak syntax with its induction principle using rewrite rules [6] in ordinary Agda.

We denote the weak syntax by Sw, components of a displayed model over the weak syntax include the following (see [11] for how to obtain the notion of displayed model for any theory).

$$\begin{aligned}
&\mathsf{Ty}^\bullet && : \mathsf{Ty}_{\mathsf{Sw}} \to \mathsf{Set}\\
&\mathsf{Tm}^\bullet && : \mathsf{Con}^\bullet\,\Gamma \to \mathsf{Ty}^\bullet\,A \to \mathsf{Tm}_{\mathsf{Sw}}\,\Gamma\,A \to \mathsf{Set}\\
&- \Rightarrow^\bullet - && : \mathsf{Ty}^\bullet\,A \to \mathsf{Ty}^\bullet\,B \to \mathsf{Ty}^\bullet\,(A \Rightarrow_{\mathsf{Sw}} B)\\
&- \cdot^\bullet - && : \mathsf{Tm}^\bullet\,\Gamma^\bullet\,(A^\bullet \Rightarrow^\bullet B^\bullet)\,t \to \mathsf{Tm}^\bullet\,\Gamma^\bullet\,A^\bullet\,u \to \mathsf{Tm}^\bullet\,\Gamma^\bullet\,A^\bullet\,(t \cdot_{\mathsf{Sw}} u)\\
&\cdot[]^\bullet && : \mathsf{transp}_{(\mathsf{Tm}^\bullet\,\Gamma^\bullet\,B^\bullet)}\,\cdot[]_{\mathsf{Sw}}\,(t^\bullet \cdot^\bullet u^\bullet)[\sigma^\bullet]^\bullet = (t^\bullet[\sigma^\bullet]^\bullet) \cdot^\bullet (u^\bullet[\sigma^\bullet]^\bullet)
\end{aligned}$$

Each displayed component is over the corresponding component in the weak syntax. In particular, the left hand side of the equation $\cdot[]^\bullet$ is transported over $\cdot[]_{\mathsf{Sw}}$ because it has type $\mathsf{Tm}^\bullet\,\Delta^\bullet\,B^\bullet\,((t \cdot_{\mathsf{Sw}} u)[\sigma]_{\mathsf{Sw}})$ while the right hand side has type $\mathsf{Tm}^\bullet\,\Delta^\bullet\,B^\bullet\,((t[\sigma]_{\mathsf{Sw}}) \cdot_{\mathsf{Sw}} (u[\sigma]_{\mathsf{Sw}}))$.

By induction on terms of the weak syntax, we define a new instantiation operation $-[-]$ mutually with its correctness property. Parts of its definition are below. Formally, this is given as interpretation into a displayed model over Sw instead of pattern matching.

$$\begin{aligned}
&-[-]_{\mathsf{new}} && : \mathsf{Tm}_{\mathsf{Sw}}\,\Gamma\,A \to \mathsf{Sub}_{\mathsf{Sw}}\,\Delta\,\Gamma \to \mathsf{Tm}_{\mathsf{Sw}}\,\Delta\,A\\
&\mathsf{correct} && : (t : \mathsf{Tm}_{\mathsf{Sw}}\,\Gamma\,A) \to t[\sigma]_{\mathsf{new}} = t[\sigma]_{\mathsf{Sw}}\\
&(t \cdot_{\mathsf{Sw}} u)[\sigma]_{\mathsf{new}} && :\equiv (t[\sigma]_{\mathsf{new}}) \cdot_{\mathsf{Sw}} (u[\sigma]_{\mathsf{new}})\\
&\mathsf{correct}\,(t \cdot_{\mathsf{Sw}} u) && :\equiv \cdot[]_{\mathsf{Sw}}
\end{aligned}$$

Now we define the *strict syntax* Ss as a new model of our theory. Most components are the same as in the weak syntax, except instantiation which is given by the above recursive definition $-[-]_{\mathsf{Ss}} :\equiv -[-]_{\mathsf{new}}$. Substitution laws in Ss are reflexivity, e.g. $\cdot[]_{\mathsf{Ss}} :\equiv \mathsf{refl}$. Calling Ss *syntax* is justified by the fact that all its operations are definitionally or (in the case of $-[-]_{\mathsf{Ss}}$) propositionally equal to those in Sw. Thus it is straightforward to derive its induction principle: every displayed model over Ss has a strict section. The notion of displayed model over Ss is slightly simpler than that over Sw because the substitution laws do not need transport anymore. E.g. $-[]^\bullet$ now has type $(t^\bullet \cdot^\bullet u^\bullet)[\sigma^\bullet]^\bullet = (t^\bullet[\sigma^\bullet]^\bullet) \cdot^\bullet (u^\bullet[\sigma^\bullet]^\bullet)$, the two sides have the same type.

Using the strict syntax is very convenient: congruence laws for substitutions are definitional, most substitutions disappear automatically. Proving canonicity using the weak syntax takes 190 lines of Agda code compared to 160 lines using the strict syntax. In a completely strict syntax (where all equations are definitional), canonicity takes 44 lines of code. This corresponds to the complete lack of transport hell, and can be achieved using rewrite rules [6] for equations in the syntax or shallow embedding [12], both of which we consider cheating. The formalisation is available online at `https://bitbucket.org/akaposi/qiirt`.

We would like to make more syntactic equalities definitional. Some are easy e.g. $\mathsf{q}[\sigma, t] = t$, some are tricky e.g. β law for function space which relies on full normalisation, and some are hopeless, e.g. the functor law $t[\sigma \circ \rho] = t[\sigma][\rho]$. The method clearly works for dependent types (proper CwFs), and we believe that it can be generalised to arbitrary languages with bindings defined as second order generalised algebraic theories [14, 3], thus obtaining first-order intrinsic syntaxes with recursive substitutions.

A general definition of quotient inductive-inductive-recursive types is still lacking. If one is able to formalise a QIIRT using pattern matching in Agda, then she should be able to first turn it into a QIIT by making the equations of the function definition propositional. Then the QIIT can be strictified by redefining the recursive operation using a displayed model just as we did above for instantiation in the syntax.

References

[1] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, 2(POPL):23:1–23:29, 2018.

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

[3] Rafaël Bocquet, Ambrus Kaposi, and Christian Sattler. For the metatheory of type theory, internal sconing is enough. *CoRR*, abs/2302.05190, 2023.

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

[5] James Chapman. Type theory should eat itself. In Andreas Abel and Christian Urban, editors, *Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice, LFMTP@LICS 2008, Pittsburgh, PA, USA, June 23, 2008*, volume 228 of *Electronic Notes in Theoretical Computer Science*, pages 21–36. Elsevier, 2008.

[6] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The taming of the rew: a type theory with computational assumptions. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[7] Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2006.

[8] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symb. Log.*, 65(2):525–549, 2000.

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

[10] Peter G. Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. Small induction recursion. In Masahito Hasegawa, editor, *Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings*, volume 7941 of *Lecture Notes in Computer Science*, pages 156–172. Springer, 2013.

[11] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.*, 3(POPL):2:1–2:24, 2019.

[12] 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.

[13] Loïc Pujet and Nicolas Tabareau. Impredicative observational equality. *Proc. ACM Program. Lang.*, 7(POPL):2171–2196, 2023.

[14] Taichi Uemura. *Abstract and concrete type theories.* PhD thesis, Institute for Logic, Language and Computation (ILLC), University of Amsterdam, 2021.

[15] 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.

12

Session 16: Foundations of type theory and constructive mathematics

Self-contained rules for classical and intuitionistic quantifiers

Herman Geuvers[1] and Tonny Hurkens

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

We introduce natural deduction rules for the well-known quantifiers $\forall$ and $\exists$ and the less known quantifiers И ("there is no x for which $\varphi(x)$") and Ɔ ("there is some x for which $\varphi(x)$ doesn't hold") that are *self-contained*. With self-contained, we mean that the rules are just about the quantifiers themselves and do not need other quantifiers or connectives to be expressed. As a by-product we have derivations of well-known classical tautologies that only involve $\forall$ and $\vee$ that satisfy the subformula property (and thus do not rely on negation and the law of excluded middle), e.g. $\forall x.(P\,x \vee C) \vdash (\forall x.P\,x) \vee C$.

Our derivation rules for the quantifiers $\forall, \exists, \text{И}, \text{Ɔ}$ are derived from the *truth table natural deduction* approach (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]. The rules come in two flavors: (1) a complete natural deduction calculus for classical predicate logic; (2) a complete natural deduction calculus for intuitionistic logic. In both cases one can choose which of the quantifiers and propositional connectives one wants to have. The rules are self-contained, so adding a specific quantifier doesn't have any prerequisites on adding other connectives first.

The main results are the following.

1. Self-contained classical deduction rules for $\forall, \exists, \text{И}, \text{Ɔ}$.
2. A Tarskian-style semantics for $\forall, \exists, \text{И}, \text{Ɔ}$ for which the classical rules are sound.
3. Classical derivations, e.g. of $\vdash \exists x.(\exists y.P\,y) \to P\,x$ and of $\forall x.(P\,x \vee C) \vdash (\forall x.P\,x) \vee C$ that satisfy the subformula property.
4. Self-contained intuitionistic deduction rules for $\forall, \exists, \text{И}, \text{Ɔ}$.
5. A Kripke semantics for $\forall, \exists, \text{И}, \text{Ɔ}$ for which the intuitionistic rules are sound.
6. Proofs (intuitionistic derivations) showing that $\neg\exists x.\varphi$ and $\text{И}x.\varphi$ and $\forall x.\neg\varphi$ are equivalent, and that $\exists x.\neg\varphi \vdash \text{Ɔ}x.\varphi$ and $\text{Ɔ}x.\varphi \vdash \neg\forall x.\varphi$, but not the other way around. The formula $\text{Ɔ}x.\varphi$ expresses that "there is a counter-example to $\varphi(x)$", and it is intuitionistically in between $\exists x.\neg\varphi$ and $\neg\forall x.\varphi$.

For (3), we have the following classical derivation of $\vdash \exists x.(\exists y.P\,y) \to P\,x$.

$$\dfrac{\dfrac{\dfrac{\exists y.P\,y \vdash \exists y.P\,y}{\exists y.P\,y \vdash P\,a_{\exists y.P\,y}}\ \exists\text{-elC}}{\vdash (\exists y.P\,y) \to P\,a_{\exists y.P\,y}}}{\vdash \exists x.(\exists y.P\,y) \to P\,x}$$

Here, $\exists$-elC is the classical $\exists$-elimination rule, which allows to conclude $P\,a_{\exists y.P\,y}$ from $\exists y.P\,y$. The $a_{\exists y.P\,y}$ is a special *witness constant* that plays the role of "the element d for which $P\,d$

holds". Syntactically, we add constants $a_{\exists x.\varphi}$ for all formulas $\varphi(x)$ (and similarly $a_{\forall x.\varphi}$, $a_{\text{И} x.\varphi}$ and $a_{\text{Ɔ} x.\varphi}$), and we only deal with closed formulas.

Intuitionistially, these witness constants act as the *eigenvariables* in rules like $\exists$-elimination and $\forall$-introduction, where there is a side-condition that a local variable should not occur in other assumptions or the conclusion. The constant $a_{\exists x.\varphi}$ will play the role of this local variable in the intuitionistic $\exists$-elimination rule (and similarly $a_{\forall x.\varphi}$ in the intuitionistic $\forall$-introduction rule, and $a_{\text{И} x.\varphi}$ and $a_{\text{Ɔ} x.\varphi}$ will play similar roles). So, intuitionistically, these witness constants don't have a special semantics, which conforms with the fact that the well-known rules for $\forall$ and $\exists$ are intuitionistic rules.

Classically, the witness constants have a specific semantics. We want to make sure that $\forall x.\psi \Leftrightarrow \psi(a_{\forall x.\psi})$, so for the interpretation in a model $\mathcal{M}$ we have

$$[\![a_{\forall x.\varphi}]\!] := \begin{cases} \text{an arbitrary element of } D & \text{if } \mathcal{M} \models \forall x.\varphi \\ \text{some element } d \text{ for which } \mathcal{M} \not\models \varphi(d) & \text{if } \mathcal{M} \not\models \forall x.\varphi. \end{cases}$$

For $a_{\exists x.\psi}$, we want to make sure that $\exists x.\psi \Leftrightarrow \psi(a_{\exists x.\psi})$, so for the interpretation in a model $\mathcal{M}$ we have

$$[\![a_{\exists x.\varphi}]\!] := \begin{cases} \text{some element } d \text{ for which } \mathcal{M} \models \varphi(d) & \text{if } \mathcal{M} \models \exists x.\varphi \\ \text{an arbitrary element of } D & \text{if } \mathcal{M} \not\models \exists x.\varphi. \end{cases}$$

Similarly, we make choices for $[\![a_{\text{И} x.\psi}]\!]$ and $[\![a_{\text{Ɔ} x.\psi}]\!]$, to make sure that $\text{И} x.\psi \Leftrightarrow \neg\psi(a_{\text{И} x.\psi})$, and $\text{Ɔ} x.\psi \Leftrightarrow \neg\psi(a_{\text{Ɔ} x.\psi})$.

We give the classical deduction rules (indicated with C) and the intuitionistic deduction rules (indicated with I) for $\forall$, $\exists$, И and Ɔ. (If nothing is indicated the rules are classical and intuitionistic.) If the rule has a "non-occurrence" side condition, we give the context Γ, otherwise we omit it.

Deduction rules for $\forall$, where t is an arbitrary term. We abbreviate $a_{\forall x.\varphi}$ to $a_\forall$.

$$\boxed{\dfrac{\vdash \forall x.\varphi}{\vdash \varphi(t)}\,\forall\text{-el} \quad\Big\|\quad \dfrac{\vdash \varphi(a_\forall)}{\vdash \forall x.\varphi}\,\forall\text{-inC} \qquad \dfrac{\Gamma \vdash \varphi(a_\forall)}{\Gamma \vdash \forall x.\varphi}\,\forall\text{-inI, if } a_\forall \notin \Gamma}$$

Deduction rules for $\exists$, where t is an arbitrary term. We abbreviate $a_{\exists x.\varphi}$ to $a_\exists$.

$$\boxed{\dfrac{\vdash \exists x.\varphi}{\vdash \varphi(a_\exists)}\,\exists\text{-elC} \qquad \dfrac{\Gamma \vdash \exists x.\varphi \quad \Gamma, \varphi(a_\exists) \vdash \psi}{\Gamma \vdash \psi}\,\exists\text{-elI, if } a_\exists \notin \Gamma, \psi \quad\Big\|\quad \dfrac{\vdash \varphi(t)}{\vdash \exists x.\varphi}\,\exists\text{-in}}$$

Deduction rules for И, where t is an arbitrary term. We abbreviate $a_{\text{И} x.\varphi}$ to $a_{\text{И}}$

$$\boxed{\dfrac{\vdash \text{И} x.\varphi \quad \vdash \varphi(t)}{\vdash \psi}\,\text{И-el} \quad\Big\|\quad \dfrac{\text{И} x.\varphi \vdash \psi \quad \varphi(a_{\text{И}}) \vdash \psi}{\vdash \psi}\,\text{И-inC} \qquad \dfrac{\Gamma, \varphi(a_{\text{И}}) \vdash \text{И} x.\varphi}{\Gamma \vdash \text{И} x.\varphi}\,\text{И-inI, if } a_{\text{И}} \notin \Gamma}$$

Deduction rules for Ɔ, where t is an arbitrary term. We abbreviate $a_{\text{Ɔ} x.\varphi}$ to $a_{\text{Ɔ}}$

$$\boxed{\begin{array}{c} \dfrac{\vdash \text{Ɔ} x.\varphi \quad \vdash \varphi(a_{\text{Ɔ}})}{\vdash \psi}\,\text{Ɔ-elC} \qquad \dfrac{\Gamma \vdash \text{Ɔ} x.\varphi \quad \Gamma \vdash \varphi(a_{\text{Ɔ}})}{\Gamma \vdash \psi}\,\text{Ɔ-elI, if } a_{\text{Ɔ}} \notin \Gamma \\ \hline\hline \dfrac{\text{Ɔ} x.\varphi \vdash \psi \quad \varphi(t) \vdash \psi}{\psi}\,\text{Ɔ-inC} \qquad \dfrac{\varphi(t) \vdash \text{Ɔ} x.\varphi}{\vdash \text{Ɔ} x.\varphi}\,\text{Ɔ-inI} \end{array}}$$

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 für Informatik.

Terms as Types: Calculations in the λ-Calculus

José Espírito Santo[1], René Gazzari[2], and Luís Pinto[3]

[1] CMAT, University of Minho, Braga, Portual
`jes@math.uminho.pt`

[2] CMAT, University of Minho, Braga, Portual
`gazzari@math.uminho.pt`

[3] CMAT, University of Minho, Braga, Portual
`luis@math.uminho.pt`

Gazzari [4] proposed the calculus of Natural Calculation (NC), an extension of Gentzen's [5] Natural Deduction (ND) by proper term rules permitting a natural representation of calculations (inside of proofs). In NC, the first order terms t are first-class members of the calculus on a par with formulae: they can be assumed and discharged; the elimination of the equality rule permits calculation steps in the calculus transforming term premises into terms conclusions; the evaluation of calculations is the corresponding introduction rule for the equality symbol. A subtlety of NC is that the elimination of equality rule is given in two polarities: while the positive version applied on an equation $t = s$ permits the replacement of some occurrences of t in a term r by the term s, the negative version "reads" the equation from the right to the left and allows, correspondingly, the replacement of s by t. NC is complete and sound with respect to the usual treatment of equality in ND and can, therefore, be seen as a natural alternative to the this treatment. Indrzejczak [7] carried over Gazzari's idea of term rules governing the equality symbol to the sequent calculus.

Recalling the Curry-Howard correspondence [6, 9] associating (simply typed) λ-calculus with ND, it is quite natural to ask, how a λ-calculus corresponding to NC would look like. A first answer to this question is a version of the λ-calculus with (equality) calculations (LCC$_=$) corresponding to the (intuitionistic) fragment of NC over the implication ($\to$), the universal quantifier ($\forall$) and the equality symbol ($=$)). Besides the usual proof terms P representing, essentially, the standard parts of NC derivations, there are also calculation terms C in LCC$_=$ for the representation of the calculations inside of the NC derivations. Proof terms and calculation terms depend on each other and are defined in parallel (see figure 1).

The types of proof terms are, as expected, first order formulae (more precisely, the conclusions of the derivations corresponding to the respective proof terms). The interesting case are the calculation terms: it turned out that the type of a calculation term C is a first order term t. Term assumptions of NC are represented by a special variable y; the term type t of the variable y is the starting term of a calculation. The term type s of an arbitrary calculation term C is the final term of the corresponding calculation, the evaluation of such a calculation from t to s (represented by the proof term $P \equiv \lambda y.C$) yields the result $t = s$, which becomes the type of the proof term P. The precise formulation of some typing rules using the terms-as-types paradigm is found in figure 2.[1] The typing refers to sequents of the forms $\Gamma;\ y : t \vdash C : s$ and $\Gamma; \vdash P : A$.

It is worth mentioning that if we analyse our types, namely the first order terms and formulae, as λ-expressions of simply typed λ-calculus (STLC) as done by Church [2], then we obtain the following chains of dependencies, where ι is the type of individuals and o the type of formulae: $C : t : \iota$ and $P : A : o$

[1] Notation: the expression $\mathtt{r}$ can be seen as a first order term r with "holes" (similar to the contexts introduced in Barendregt [1], which are λ-terms with holes). In $\mathtt{r}[t]$ the holes are "filled" with the first order term t, in $\mathtt{r}[s]$ with s. This way, the replacement of some occurrences of t in r by s can be given precisely. Gazzari [3] provides a detailed analysis of this approach to the notions of positions and occurrences.

Figure 1: Grammar of LCC$_=$

(first order terms)	t, s	$::=$	$v \mid c \mid f(t_0, \ldots t_n)$
(formulae)	A, B	$::=$	$t = s \mid A \to B \mid \forall v.A$
(calculation terms)	C, D	$::=$	$y \mid P^{\pm}C$
(proof terms)	P, Q	$::=$	$x \mid \lambda y.C \mid \lambda x.P \mid PQ \mid \lambda v.P \mid Pt$

Figure 2: Some Typing Rules for LCC$_=$

$$\frac{\Gamma;\ y : t \vdash C : s}{\Gamma;\ \vdash (\lambda y.C) : t = s}\ (I_=) \quad ; \quad \frac{\Gamma;\ \vdash P : s_0 = s_1 \qquad \Gamma;\ y : t \vdash C : \mathtt{r}[s_0]}{\Gamma;\ y : t \vdash P^+C : \mathtt{r}[s_1]}\ (E^{+}_{=}) \quad ;$$

$$\frac{\Gamma;\ \vdash P : s_0 = s_1 \qquad \Gamma;\ y : t \vdash C : \mathtt{r}[s_1]}{\Gamma;\ y : t \vdash P^-C : \mathtt{r}[s_0]}\ (E^{-}_{=})$$

LCC$_=$ is equipped with the standard reduction rules $\beta_\to$ and $\beta_\forall$ expressing the elimination of maximal formulae of the forms $A \to B$ and $\forall v.A$, respectively. In addition, it includes the two rules

$$(\lambda y.C)^+D \ \rhd\ [D/y]C \qquad\qquad (\lambda y.C)^-D \ \rhd\ [D/y]\overline{C}$$

where $\overline{C}$, the *dual* calculation of C, is defined by recursion on C as $\overline{y} \equiv y$ and $\overline{P^{\pm}C} \equiv [P^{\mp}y/y]\overline{C}$.

As the special variable y does not occur free in proof terms, and has a single free occurrence in C, we can understand y as the the end-marker of a list of proof terms: $C \equiv P_1^{\pm}(\cdots(P_n^{\pm}y)\cdots)$. Hence calculation terms are a form of lists of proof terms, where each P_i may encapsulate another calculation term, when P_i has the form $\lambda y.C'$. In this reading, y is the empty list, $\lambda y.C$ may be written as a capsule $< C >$, $\overline{C}$ is the reverse of C, and $[D/y]C$ the concatenation of C with D, denoted $C; D$. Such lists β-reduce, as follows: $< C >^{\pm} D \to (\pm C); D$, where $+C \equiv C$ and $-C \equiv \overline{C}$.

Our investigations of LCC$_=$ are work in progress. As a first result, we communicate strong normalisation, which is established by using a translation into STLC and lifting strong normalisation this way from STLC into LCC$_=$. The translation incorporates some different aspects: (1) all terms (as types) are mapped to a single variable p (permitting the representation of the undirected equality by the directed arrow), (2) the complexity of the translation of formulae (as types) has to be lifted accordingly, (3) permutations of calculation terms due to β-reductions are anticipated by the translation, which allows (4) to translate positive and negative applications of calculation terms into the unpolarised applications of STLC. Finally, (5) all traces of the representation of the universal quantifier (in the proof terms) are eliminated. (The latter idea is found in Sørensen and Urzyczyn [9, p. 219].)

We expect to prove as next result confluence of LCC$_=$. Future work includes the extension of LCC$_=$ to permit the treatment of relation symbols R (different from the equality symbol). In particular, a substitution rule permitting the replacement of equal terms in relational formulae (a rule of NC not considered yet) has to be represented in LCC$_=$. Another interesting line of investigation is the development of a calculus LCC$_<$, where the fundamental calculations are smaller-than calculations and where the equality becomes a defined concept (analogously to the biimplication defined in terms of the implication).

References

[1] H. Barendregt. *The lambda calculus.* North-Holland, 1981.

[2] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[3] R. Gazzari. *Formal Theories of Occurrences and Substitutions.* PhD thesis, University of Tübingen, 2020.

[4] R. Gazzari. The Calculus of Natural Calculation. *Studia Logica*, 109:1375–1411, 2021.

[5] G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 1934.

[6] W. Howard. *The formulae-as-types notion of construction*, pages 479–490. 1980. In [8].

[7] A. Indrzejczak. A Novel Approach to Equality. *Synthese*, 199(1):4749–4774, 2021.

[8] J. P. Seldin and J. R. Hindley. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

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

Monadic Realizability for Intuitionistic Higher-Order Logic

Liron Cohen[1]*, Ariel Grunfeld[1]*, and Ross Tate

Ben-Gurion University, Israel

Abstract

The standard construction for realizability semantics of intuitionistic higher-order logic is based on partial combinatory algebras as an abstract computation model with a single computational effect, namely, non-termination. Many computational effects can be modelled using monads, where programs are interpreted as morphisms in the corresponding Kleisli category. To account for a more general notion of computational effects, we here construct effectful realizability models via evidenced frames, where the underlying computational model is defined in terms of an arbitrary monad. Concretely, we generalize partial combinatory algebras to combinatory algebras over a monad and use monotonic post-modules to relate predicates to computations.

Evidenced Frames (EF) provide a general framework for constructing realizability triposes, including various computational effects that go beyond partial computation [4]. An evidenced frame is a tagged variant of complete Heyting prealgebras, where instead of the binary preorder relation $\varphi \leq \psi$ we have a ternary evidence relation $\varphi \xrightarrow{e} \psi$ where e is considered as evidence for the judgment $\varphi \leq \psi$. Similarly, each of the components of a complete Heyting prealgebra (reflexivity, transitivity, top, conjunction, implication, and universal quantification) are defined in terms of their evidence.

The standard construction of realizability models for intuitionistic higher-order logic (**iHOL**) interprets formulas as functions to subsets of the set of codes in a partial combinatory algebra (PCA), constructing a tripos (called "the realizability tripos") by using the codes as evidence for the validity of entailments, and using the functional completeness of the PCA to construct specific codes to realize the logical constants of **iHOL**. Evidenced frames can similarly be used to construct a tripos, in a manner that separates the realizability construction into two phases: first, constructing an EF from a PCA, and then constructing a tripos from the EF. This separation gives us a single structure that explicitly relates the logical content to the computational content, and allows us to replace the PCA with other viable models, such as relational combinatory algebras (RCAs) and stateful combinatory algebras (SCAs), as described in [3].

The main goal of this work is to generalize these results further by abstracting the details of the specific computational effects, and instead relying on the ideas first introduced in [9] where the effects are encapsulated behind some arbitrary monad. For this, RCA and SCA can be considered as special cases of a more general notion: Monadic Combinatory Algebra (MCA).

Definition 1 (Monadic Combinatory Algebra)**.** *Given a strong monad $T : \mathbb{C} \to \mathbb{C}$, a* Monadic Applicative Structure *(MAS) is an object of "codes" $\mathbb{A}$ together with an application Kleisli morphism: $\alpha \in \mathbb{C}(\mathbb{A} \times \mathbb{A}, T\mathbb{A})$. We say that $\mathbb{A}$ is a* Monadic Combinatory Algebra *(MCA) when $\mathbb{A}$ is a Turing object [2] in the Kleisli category of T, considered as a restriction category.*

When $\mathbb{C} = \mathbf{Set}$, the definition coincides with a PCA for T the sub-singleton monad, with an RCA for the power-set monad, and an SCA for the (increasing) state power-set monad.

In **Set**, an MCA can be more explicitly defined using an abstraction operator over terms. Given a *MAS* $\mathbb{A}$, the set $E_n(\mathbb{A})$ of *terms over* $\mathbb{A}$ is defined by the grammar:

$$e \ ::= \ 0 \mid \ \dots \ \mid n-1 \mid c \in \mathbb{A} \mid e \bullet e$$

*This research is supported by Grant No. 2020145 from the United States-Israel Binational Science Foundation (BSF)

$E_0(\mathbb{A})$ is the set of *closed terms over* $\mathbb{A}$, and $e[c]$ denotes the straightforward *substitution*. *Evaluation* $\nu : E_0(\mathbb{A}) \to T(\mathbb{A})$ is defined by induction on $E_0(\mathbb{A})$ (using do-notation):

$$\nu(c) := \eta(c) \qquad \nu(e_f \bullet e_a) := \text{do } c_f \leftarrow \nu(e_f) \ ; \ c_a \leftarrow \nu(e_a) \ ; \ \alpha(c_f, c_a)$$

Proposition 1 (Monadic Combinatory Algebra)**.** *Given a MAS* $\mathbb{A}$*,* $\mathbb{A}$ *is an MCA if for each* $n \in \mathbb{N}$ *there's an abstraction operator* $\langle \lambda^n. (-) \rangle : E_{n+1}(\mathbb{A}) \to \mathbb{A}$ *s.t:* $\langle \lambda^{n+1}.e \rangle \cdot c = \eta(\langle \lambda^n.e[c] \rangle)$ *and* $\langle \lambda^0.e \rangle \cdot c = \nu(e[c])$*.*

MCAs allow us to construct evidence for entailments in a similar manner to PCAs. For PCAs, we consider predicates over codes and say $\varphi \xrightarrow{e} \psi$ whenever the predicate ψ is satisfied by the output of the application of e on an input which satisfies φ. In MCAs, while the input of the application is a code, the output is wrapped inside T, so to relate a predicate over codes to a predicate over wrapped codes, we need to use a *post-module* to "lift" predicates on $\mathbb{C}$ to predicates on the Kleisli category of T.

Definition 2 (post-module)**.** *Given a monad* T*, a* post-module *is a tuple* $(\mathbb{P}, P, \rho)$ *where* $\mathbb{P}$ *is a category,* $P : \mathbb{C} \to \mathbb{P}$ *is a functor, and* $\rho : PT \Rightarrow P$ *is a natural transformation for which* $\rho \circ P\eta = id_P$ *and* $\rho \circ P\mu = \rho \circ \rho_T$*.*

When $\mathbb{P} = \mathbf{Prost}^{\mathrm{op}}$ (the dual of the category of preordered sets), for any morphism f in $\mathbb{C}$, the function Pf has to be a monotonic, and similarly each component $\rho_X : PX \to PTX$ of ρ has to be monotonic. So when $\mathbb{P} = \mathbf{Prost}^{\mathrm{op}}$ we use the term *monotonic post-module.*

Proposition 2. *Given a post-module* $(\mathbb{P}, P, \rho)$*, we get a functor* $P_\rho : \mathbb{C}_T \to \mathbb{P}$ *where* $\mathbb{C}_T$ *is the Kleisli category of* T *on* $\mathbb{C}$*.* P_ρ *is defined on objects by* $P_\rho X = PX$ *and on a morphism* $f \in \mathbb{C}_T(A, B)$ *by* $P_\rho f = \rho_B \circ Pf$*.*

P_ρ can be considered as a categorical counterpart of Dijkstra's weakest precondition transformer [5] Given an MCA $\mathbb{A}$ and a *monotonic post-module* $(\mathbf{Prost}^{\mathrm{op}}, P, \rho)$ we can define an evidence relation on predicates on $\mathbb{A}$. If $\varphi, \psi \in P(\mathbb{A})$ and $e \in \mathbb{C}(\mathbf{1}, \mathbb{A})$ we say that $\varphi \xrightarrow{e} \psi$ whenever $\varphi \leq P_\rho(\alpha \circ \langle e \circ !, \mathrm{id}_{\mathbb{A}} \rangle)(\psi)$ (where ! is the unique morphism to the terminal object).

An interesting special case is when $\mathbb{C} = \mathbf{Set}$. In $\mathbf{Set}$, whenever Ω is a preordered set, we have the **Proset** functor $PA = A \to \Omega$, with $Pf(\varphi) = \varphi \circ f$. This functor can become a monotonic post-module by using a *monotonic* T*-algebra* [1] $\omega : T\Omega \to \Omega$, yielding $\rho_A(\varphi) = \omega \circ T\varphi$ (for $\varphi : A \to \Omega$). Equivalently, we can use a monad morphism into the monotone continuation monad, as described in [8]. We already have a construction (formalized in Coq) of an EF based on it. While capturing PCAs, RCAs, and SCAs, it seems not general enough to encompass other interesting EFs, such as EFs of probabilistic computation, which would probably require T to be the Giry monad [6]. Furthermore, it does not seem to encompass interesting post-modules for the continuation monad.

To get a more bird's-eye view of the necessary components needed to construct an EF, we define a deductive system called **EffHOL** which extends **iHOL** with *effectful* terms along with the operations: `return` to turn pure terms into effectful terms, `bind` to compose effectful terms, and `after` to relate effectful terms with formulas. The `after` operation acts as a quantifier that takes an effectful term e and a formula φ and returns the formula `after` $x := e \,.\, \varphi$, denoting that after the computation of e, the formula φ holds. To relate **EffHOL** to **iHOL**, we require **EffHOL** to have a special type of "codes" $\mathbb{A}$, denoting programs in an untyped programming language. $\mathbb{A}$ is required to have an effectful "application" operator `ap` which takes two pure codes e and c and returns an effectful term `ap` (e, c), corresponding to the application of the program denoted by e to the input denoted by c, yielding a computation (hence the result is an effectful term). By combining `ap` with `after` we can use **EffHOL** to describe Hoare triples $\{\varphi\}\ e\ \{\psi\}$ [7]:

$$c_a \mid \varphi[c_a] \ \vdash \ \texttt{after}\, c_r := \texttt{ap}(e, c_a) \ .\ \psi[c_r]$$

References

[1] Alejandro Aguirre and Shin-ya Katsumata. Weakest preconditions in fibrations. *Electronic Notes in Theoretical Computer Science*, 352:5–27, 2020.

[2] J Robin B Cockett and Pieter JW Hofstra. Introduction to turing categories. *Annals of pure and applied logic*, 156(2-3):183–209, 2008.

[3] Liron Cohen, Sofia Abreu Faro, and Ross Tate. The effects of effects on constructivism. *Electronic Notes in Theoretical Computer Science*, 347:87–120, 2019.

[4] Liron Cohen, Étienne Miquey, and Ross Tate. Evidenced frames: a unifying framework broadening realizability models. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2021.

[5] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[6] Michele Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis: Proceedings of an International Conference Held at Carleton University, Ottawa, August 11–15, 1981*, pages 68–85. Springer, 2006.

[7] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[8] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *arXiv preprint arXiv:1903.01237*, 2019.

[9] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

13

Session 17: Links between type theory and functional programming

Operations On Syntax Should Not Inspect The Scope

Jesper Cockx

TU Delft, Delft, Netherlands

When implementing or formalizing the syntax of a language with names and binders, one challenging task is establishing and preserving well-scopedness. This is especially true when implementing a dependent type checker, where types bind variables and terms with free variables are evaluated. Luckily, if we implement this type checker itself in a dependently typed language, we can work with *well-scoped syntax*, i.e. syntax that is statically known to be well-scoped by the type system. For example, here is a minimal definition of well-scoped syntax for the untyped lambda calculus in Agda:

```
data Var : (n : ℕ) → Set where
  zero : Var (suc n)
  suc  : Var n → Var (suc n)
```

```
data Term (n : ℕ) : Set where
  var : Var n → Term n
  lam : Term (suc n) → Term n
  app : Term n → Term n → Term n
```

Brady et al. [2003] have taught us that inductive families such as Var and Term need not store their indices: the number n can be safely erased during compilation. However, to produce efficient compiled code we should also ensure that operations on the syntax do not inspect the scope at run-time. In a language with support for runtime irrelevance [McBride, 2016, Atkey, 2018] such as Idris 2 or Agda, we can enforce this property statically. But this reveals a problem: to implement a function right : Var n → Var $(k + n)$ that weakens a variable by adding k unused variables to the scope, it must apply the suc constructor k times to its argument, so erasing k is impossible! This example shows that using ℕ as the type of scopes does not work.

This leads us to the question: is it possible to design types Scope : Set and Var : Scope → Set such that all necessary operations on variables can be defined without inspecting the scope. To make this question more concrete, let me list some operations that I consider 'necessary':

1. **Decidable equality** of variables: $\_\overset{?}{=}\_ : (x\ y : \mathsf{Var}\ \alpha) \to \mathsf{Dec}\ (x \equiv y)$.
2. An **empty scope** $\circ$: Scope such that $\mathsf{Var}\ \circ \simeq \bot$.
3. A **singleton scope** $\bullet$: Scope such that $\mathsf{Var}\ \bullet \simeq \top$.
4. A **disjoint union** $\_\diamond\_$: Scope → Scope → Scope such that $\mathsf{Var}\ (\alpha \diamond \beta) \simeq \mathsf{Var}\ \alpha \uplus \mathsf{Var}\ \beta$.
5. A **weakening** coerce : $\alpha \subseteq \beta \to \mathsf{Var}\ \alpha \to \mathsf{Var}\ \beta$, where $\_\subseteq\_$: Scope → Scope → Set is a preorder on scopes.
6. For any $p : \alpha \subseteq \beta$, a **complement** p^C : Scope such that $p^C \subseteq \beta$ and $p^C \subseteq (\mathsf{trans}\ p\ q)^C$ for any $q : \beta \subseteq \gamma$.

Instead of using ℕ, let us represent scopes as *binary trees* where each leaf is either an empty scope ∘ or a singleton •:

```
data Scope : Set where
  ∘ •  : Scope
  _◇_  : Scope → Scope → Scope
```

Rather than define Var and _⊆_ directly, we can define both in terms of a *proof relevant separation algebra* [Rouvoet et al., 2020], a ternary relation on scopes that determines how the names in the third scope are distributed over the first two.

```
data _⋈_≡_ : (α β γ : Scope) → Set where
  ∘-l   : ∘ ⋈ β ≡ β
  ∘-r   : α ⋈ ∘ ≡ α
  join  : α ⋈ β ≡ (α ⋄ β)
  swap  : α ⋈ β ≡ (β ⋄ α)
  ⋄-ll  : (α₂ ⋈ β  ≡ δ) → (α₁ ⋈ δ  ≡ γ) → (α₁ ⋄ α₂) ⋈ β          ≡ γ
  ⋄-lr  : (α₁ ⋈ β  ≡ δ) → (δ  ⋈ α₂ ≡ γ) → (α₁ ⋄ α₂) ⋈ β          ≡ γ
  ⋄-rl  : (α  ⋈ β₂ ≡ δ) → (β₁ ⋈ δ  ≡ γ) → α          ⋈ (β₁ ⋄ β₂) ≡ γ
  ⋄-rr  : (α  ⋈ β₁ ≡ δ) → (δ  ⋈ β₂ ≡ γ) → α          ⋈ (β₁ ⋄ β₂) ≡ γ
```

Subscoping and variables can then be defined in terms of separation:

```
α ⊆ β = Σ (Erased Scope) (λ ([ γ ]) → α ⋈ γ ≡ β)
Var α  = • ⊆ α
```

Here, Erased A is a record type with constructor [_] : @0 A → Erased A. This definition of _⊆_ makes it trivial to define the complement operation $\_^{C}$, since it is just the first projection of the subscope proof.

An implementation of the operations listed above can be found at `https://github.com/jespercockx/scopes-n-roses`. Compared to the code here, it follows Pouillard [2012] by providing an abstract interface for working with scopes and support for *named* variables.

There are at least two still unresolved problems with this scope representation. The first one is that separation proofs are not unique. In particular, we can map any proof of $(\alpha_1 \diamond \alpha_2) \bowtie \beta \equiv \gamma$ to another distinct proof of the same type:

```
enlarge : (α₁ ⋄ α₂) ⋈ β ≡ γ → (α₁ ⋄ α₂) ⋈ β ≡ γ
enlarge p = ⋄-ll join (⋄-rr join p)
```

As a result, the functions Var • → ⊤ and Var $(\alpha \diamond \beta)$ → Var α ⊎ Var β are only retractions rather than equivalences.

The second problem is that introduction of scope separation makes additional operations hard or impossible to implement, such as the following property that we would like to have in addition to the six above:

7. For two separations $p : \alpha_1 \bowtie \alpha_2 \equiv \gamma$ and $q : \beta_1 \bowtie \beta_2 \equiv \gamma$ of the same scope γ, a **four-way separation** into scopes γ_1, γ_2, γ_3, and γ_4 such that $\gamma_1 \bowtie \gamma_2 \equiv \alpha_1$, $\gamma_3 \bowtie \gamma_4 \equiv \alpha_2$, $\gamma_1 \bowtie \gamma_3 \equiv \beta_1$, and $\gamma_2 \bowtie \gamma_4 \equiv \beta_2$.

To address these problems, it may be necessary still to switch to a different representation of scopes or scope representations. However, at the moment is is not even clear whether such a representation even exists. This leads us to the following question: *is possible to give an implementation of scopes and scope separation that satisfies all the properties 1-7, while keeping the size of separation proofs bounded by the size of the scopes?* While the representation of scopes presented here does not yet answer this question, the interface it offers provides new insight into the kind of properties we can enforce by using dependent and quantitative types. It is thus a first step towards an unexplored and exciting world of new variable representations.

References

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. doi: 10.1145/3209108.3209189. URL http://doi.acm.org/10.1145/3209108.3209189.

Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2003. ISBN 3-540-22164-6. URL http://springerlink.metapress.com/openurl.asp?genre=article&amp;issn=0302-9743&amp;volume=3085&amp;spage=115.

Conor McBride. I got plenty o' nuttin'. In Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella, editors, *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *Lecture Notes in Computer Science*, pages 207–233. Springer, 2016. ISBN 978-3-319-30935-4. doi: 10.1007/978-3-319-30936-1_12. URL http://dx.doi.org/10.1007/978-3-319-30936-1_12.

Nicolas Pouillard. *Namely, Painless: A unifying approach to safe programming with first-order syntax with binders. (Une approche unifiante pour programmer sûrement avec de la syntaxe du premier ordre contenant des lieurs)*. PhD thesis, Paris Diderot University, France, 2012. URL https://tel.archives-ouvertes.fr/tel-00759059.

Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. 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 284–298. ACM, 2020. ISBN 978-1-4503-7097-4. doi: 10.1145/3372885.3373818. URL https://doi.org/10.1145/3372885.3373818.

Partiality Wrecks GADTs' Functoriality

Pierre Cagne and Patricia Johann

Appalachian State University, Boone NC, USA
`cagnep@appstate.edu`, `johannp@appstate.edu`

Generalized Algebraic Data Types (GADTs) are, as their name suggests, syntactic generalizations of standard algebraic data types (ADTs) such as list, trees, etc. ADTs are known to support an initial algebra semantics (IAS) in any category with enough structure [MA86]. This IAS gives a semantic justification for the syntactic tools ADTs come with: pattern-matching, recursion rules, induction rules, etc. One of the fundamental properties of an IAS is that the interpretation of the type constructor defined by an ADT can be extended to a functor whose action on morphisms interprets the ADT's syntactic map function.

It is natural to explore a potential generalization of IAS to GADTs. However, mapping functions over elements of GADTs is notorious for being only partially defined [JG08, JC22, JP19]. This compels us to seek the potential generalized semantics in categories with an inherent notion of partiality.

In this work, we first define a categorical framework which captures a notion of partiality that is computationally relevant. We consider the main feature of computationally relevant partiality to be that functions propagate undefinedness. This is akin to how functions that are strict in the sense of [BHA86] behave. Next, we show that any semantics in this framework is trivial if we insist that the interpretations of the type constructors defined by GADTs must extend to functors.

Given a category $\mathcal{C}$, we write $\mathsf{Mor}(\mathcal{C})$ for its (possibily large) set of morphisms. Let us start by recalling a classic definition that categorifies the notion of ideal in monoids.

Definition 1. A *cosieve* in a category $\mathcal{C}$ is a (possibly large) subset $S \subseteq \mathsf{Mor}(\mathcal{C})$ such that for all morphisms $f : A \to B$ and $g : B \to C$ in $\mathcal{C}$, if $f \in S$ then $gf \in S$.

Recall that a *wide subcategory* of a category $\mathcal{C}$ is a subcategory of $\mathcal{C}$ that contains all objects of $\mathcal{C}$ (and thus all identity morphisms as well). Given any subcategory $\mathcal{D}$ of $\mathcal{C}$, we denote $\overline{\mathcal{D}}$ for its *complement*, i.e., for the (possibly large) set $\mathsf{Mor}(\mathcal{C}) \setminus \mathsf{Mor}(\mathcal{D})$.

Definition 2. A *structure of computational partiality* on a category $\mathcal{C}$ is a wide subcategory whose complement is a cosieve.

In a category $\mathcal{C}$ equipped with a structure of computational partiality $\mathcal{D}$, we call morphisms of $\mathcal{D}$ *total* and those of $\overline{\mathcal{D}}$ *properly partial*. The intuition behind Definition 2 is that $\overline{\mathcal{D}}$ is the collection of partial computations. Following that intuition, both identities and compositions of total functions must be total functions, and, when a function yields an error on an input there is no way to come back from the error by postcomposing with another function. Other categorical frameworks capturing partiality include p-categories [RR88], (bi)categories of partial maps [Car87], categories of partial morphisms [CO89], and restriction categories [CL02]. These all give rise to structures of computational partiality.

Lemma 3. *In a category equipped with a structure of computational partiality, split monomorphisms are always total.*

Proof. Let s be a split monomorphism in a category $\mathcal{C}$. Then there exists r such that $rs = \mathrm{id}$. If s were properly partial in a structure of computational partiality on $\mathcal{C}$, then rs, and thus id, would be properly partial as well. But id is total by definition, so it cannot be. Thus, s is total. □

Now fix a category $\mathcal{C}$ equipped with a structure of computational partiality $\mathcal{D}$. Suppose $\mathcal{D}$ has finite products, and write 1 for the terminal object of $\mathcal{D}$. An *interpretation* $[\![\_]\!]$ *of (a language with) GADTs in* $(\mathcal{C}, \mathcal{D})$ maps each closed type τ of the language to an object $[\![\tau]\!]$ of $\mathcal{C}$, and each function $f : \tau_1 \to \tau_2 \to \ldots \to \tau_n \to \tau$ to a total morphism $[\![f]\!] : [\![\tau_1]\!] \times [\![\tau_2]\!] \times \cdots \times [\![\tau_n]\!] \to [\![\tau]\!]$ in $\mathcal{D}$. We require that $[\![\_]\!]$ maps the unit type $\top$ to 1 and compositions of syntactic functions to the compositions of their interprations in $\mathcal{C}$. Given a n-ary GADT `G`, a functor $G : \mathcal{C}^n \to \mathcal{C}$ *manifests* `G` *relative to* $[\![\_]\!]$ if the action of G on every object $([\![\tau_1]\!], \ldots, [\![\tau_n]\!])$ is precisely $[\![\mathtt{G}\, \tau_1 \ldots \tau_n]\!]$.

Theorem 4. *Let $\mathcal{C}$ be a category equipped with structure of computational partiality $\mathcal{D}$. Suppose $[\![\_]\!]$ is an interpretation of GADTs in $(\mathcal{C}, \mathcal{D})$ relative to which each GADT can be manifested by a functor. Then $[\![\tau]\!] \simeq 1$ for all non-empty closed types τ.*

Proof. Among the GADTs in our language, we have

```
data Equal :: * → * → * where
  Refl :: ∀ α. Equal α α
```

We can use the recursion rule of GADTs to define:

```
trp :: ∀ {α β}. Equal α β → α → β
trp Refl x = x

trp⁻¹ :: ∀ {α β}. Equal α β → β → α
trp⁻¹ Refl y = y
```

Instantiating α and β to the closed types τ_1 and τ_2, respectively, the anonymous function λ `x` $\to$ $\mathtt{trp}^{-1}$ `Refl (trp Refl x)` reduces to the identity function on τ_1. By the uniqueness property of functions defined by recursion on GADTs, λ `p` $\to$ `(`λ `x` $\to$ $\mathtt{trp}^{-1}$ `p (trp p x))` reduces to the identity on τ_1 for any input `p`. Semantically this translates to the following composition being $\mathrm{id}_{[\![\tau_1]\!]}$ for any morphism $p : 1 \to [\![\mathtt{Equal}\ \tau_1\ \tau_2]\!]$ in $\mathcal{D}$:

$$[\![\mathtt{trp}^{-1}\ \{\alpha = \tau_1\}\ \{\beta = \tau_2\}]\!] \circ (p \times \mathrm{id}_{[\![\tau_2]\!]}) \circ \varphi_{[\![\tau_2]\!]} \circ [\![\mathtt{trp}\ \{\alpha = \tau_1\}\ \{\beta = \tau_2\}]\!] \circ (p \times \mathrm{id}_{[\![\tau_1]\!]}) \circ \varphi_{[\![\tau_1]\!]}$$

Here, φ_X is the canonical isomorphism $X \simeq 1 \times X$.

Now let τ be a non-empty closed type and t be a closed term of type τ. We abuse notation and write $[\![t]\!] : 1 \to [\![\tau]\!]$ for the morphism $[\![\lambda\ \_ \to t]\!]$ in $\mathcal{D}$. Since every morphism with domain 1 in $\mathcal{D}$ is a split monomorphism, so is $[\![t]\!]$. Since there exists a functor $[\![\mathtt{Equal}]\!] : \mathcal{C}^2 \to \mathcal{C}$ manifesting `Equal` relative to $[\![\_]\!]$, and since split monomorphisms are preserved by all functors, $[\![\mathtt{Equal}]\!]([\![t]\!], \mathrm{id}_1)$ is a split monomorphism as well. By Lemma 3, $[\![\mathtt{Equal}]\!]([\![t]\!], \mathrm{id}_1)$ is a morphism in $\mathcal{D}$ from $[\![\mathtt{Equal}\ \top\ \top]\!]$ to $[\![\mathtt{Equal}\ \tau\ \top]\!]$. Consider the following morphisms in $\mathcal{D}$:

$$\begin{aligned} s &= [\![\mathtt{trp}^{-1}\ \{\alpha = \tau\}\ \{\beta = \top\}]\!] \circ (p \times \mathrm{id}_1) \circ \varphi_1 && : 1 \to [\![\tau]\!] \\ r &= [\![\mathtt{trp}\ \{\alpha = \tau\}\ \{\beta = \top\}]\!] \circ (p \times \mathrm{id}_{[\![\tau]\!]}) \circ \varphi_{[\![\tau]\!]} && : [\![\tau]\!] \to 1 \end{aligned}$$

The observation at the end of the previous paragraph instantiated with $\tau_1 = \tau$, $\tau_2 = 1$ and p being the morphism $[\![\mathtt{Equal}]\!]([\![t]\!], \mathrm{id}_1) \circ [\![\mathtt{Refl}\ \{\alpha = \top\}]\!]$ in $\mathcal{D}$ shows that $sr = \mathrm{id}_{[\![\tau]\!]}$. The composition rs is necessarily id_1 because it is in $\mathcal{D}$ (i.e., is total), and 1 is terminal in $\mathcal{D}$. This explicitly gives the isomorphism announced in the statement of the theorem. □

The result holds in particular for $\mathcal{D} = \mathcal{C}$. In this case, it proves that any naive extension of IAS for ADTs to GADTs that interprets GADTs as functors on $\mathcal{C}$ directly must be trivial.

References

[BHA86] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[Car87] A. Carboni. Bicategories of partial maps. *Cahiers de topologie et géométrie différentielle catégoriques*, 28(2):111–126, 1987.

[CL02] J. R. B. Cockett and Stephen Lack. Restriction categories I: categories of partial maps. *Theoretical Computer Science*, 270(1–2):223–259, January 2002.

[CO89] Pierre-Louis Curien and A. Obtułowicz. Partiality, cartesian closedness, and toposes. *Information and Computation*, 270(1):50–95, January 1989.

[JC22] Patricia Johann and Pierre Cagne. Characterizing Functions Mappable over GADTs. In Ilya Sergey, editor, *Programming Languages and Systems*, pages 135–154, Cham, 2022. Springer Nature Switzerland.

[JG08] Patricia Johann and Neil Ghani. Foundations for structured programming with GADTs. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 43 of *POPL '08*, page 297–308, New York, NY, USA, 2008. Association for Computing Machinery.

[JP19] Patricia Johann and Andrew Polonsky. Higher-Kinded Data Types: Syntax and Semantics. In *34th Annual Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2019.

[MA86] Ernest G. Manes and Michael A. Arbib. *Algebraic Approaches to Program Semantics.* Monographs in Computer Science. Springer New York, 1986.

[RR88] E. Robinson and G. Rosolini. Categories of Partial Maps. *Information and Computation*, 79(2):95–130, November 1988.

Monadic Intersection Types

Francesco Gavazzo[1], Riccardo Treglia[2], and Gabriele Vanoni[3]

[1] Università di Pisa - `francesco.gavazzo@unipi.it`
[2] Università di Bologna - `riccardo.treglia@unibo.it`
[3] Inria - `vanonigabriele@gmail.com`

Introduction. *Intersection Types* (IT) were introduced in [CD78] to overcome the limitations of Curry's type discipline and enlarge the class of terms that can be typed. This is reached by means of the *intersection* type constructor. This way, one can assign a finite set of types to a term, thus providing a form of finite polymorphism. Intersection types *characterize* termination, that is, they type *all* terminating λ-terms. Additionally, IT have shown to be remarkably flexible, since different termination forms can be characterized by tuning details of the type system. IT have been mostly developed in the realm of the pure λ-calculus, notable exceptions being [BDL18, DLFRDR21] featuring probabilistic choice, and [DP00, DCRDR07, dLT21a, dLT21b] featuring state/references. However, current programming languages are deeply *effectful*, raising exceptions, performing input/output operations, sampling from distributions, etc. Reasoning about effectful programs becomes a challenging goal since their behaviour becomes highly interactive, depending on the external environment.

The leading question we try to answer is: *can IT be scaled up in the case of effectful λ-calculi, in a modular way?* We answer this question in the affirmative by developing a general *monadic intersection type system* for a computational λ-calculus [Mog91] with algebraic operations *à la* Plotkin and Power [PP01]. To achieve this result, we combine state-of-the-art techniques in monadic semantics, intersection types, and relational reasoning, in a novel and nontrivial way.

The Type System. The target calculus of our work is an effectful extension of the call-by-value λ-calculus with effect triggering operations $\mathtt{op}$ taken from a signature Σ:

$$\textsc{Val.}\quad \mathbb{V} \ni v, w ::= x \mid \lambda x.t \qquad\qquad \textsc{Comp.}\quad \mathbb{C} \ni t, u ::= v \mid vt \mid \mathtt{op}(t_1, \dots, t_n)$$

We can give operational semantics to this calculus in monadic style [PP01, GF21]. In particular, reduction is a (monadic) function $\mapsto : \mathbb{C} \to T(\mathbb{C})$, where $(T, \eta, \gg\!=)$ is a monad. The intersection type system, parametric in the underlying monad T, is designed in such a way that not only terms, but also types are monadic. From the informal call-by-value translation of intuitionistic logic into linear logic combined with Moggi's translation

$$A \to B \cong\ !A \multimap T(!B)$$

one can derive the following grammar for types:

$$\begin{array}{rlll} \textsc{Value Types} & \mathbb{A} \ni A & ::= & I \to M \\ \textsc{Intersection Types} & \mathbb{I} \ni I & ::= & \{A_1, \dots, A_n\} \qquad n \geq 0 \\ \textsc{Monadic Types} & \mathbb{M} \ni M, N & ::= & T(\mathbb{I}) \end{array}$$

The type assignment system in Fig. 1 is given by a relation $\vdash : \mathbb{C} \nrightarrow T(\mathbb{I})$ between terms and types, hence leaving to a situation similar to the one of operational semantics. The analogy is no coincidence: as we obtain monadic operational semantics relying on the theory of monadic relations, the very same theory allows us to define monadic type system, a monadic typing relation associating terms with *monadic types*. Differently from the pure setting, in which intersection types characterize termination of programs, in the effectful setting we would like to characterize *all* the effects produced during the evaluation via the type system.

$$\frac{A \in I}{\Gamma, x : I \vdash x : A}\ \text{VAR} \qquad \frac{[\Gamma \vdash v : I_i \to M_i]_{1\leq i\leq n} \quad \Gamma \vdash t : N \quad \mathsf{supp}(N) \subseteq \{I_1, ..., I_n\}}{\Gamma \vdash vt : N \gg\!= (I_i \mapsto M_i)}\ \text{APP}$$

$$\frac{\Gamma, x : I \vdash t : M}{\Gamma \vdash \lambda x.t : I \to M}\ \text{ABS} \qquad \frac{[\Gamma \vdash t_i : M_i]_{1\leq i\leq n}}{\Gamma \vdash \mathsf{op}(t_1, \ldots, t_n) : g_{\mathsf{op}}(M_1, \ldots, M_n)}\ \text{OP}$$

$$\frac{[\Gamma \vdash v : A_i]_{i\in F}}{\Gamma \vdash v : \{A_i\}_{i\in F}}\ \text{INT} \qquad \frac{\Gamma \vdash v : I}{\Gamma \vdash v : \eta(I)}\ \text{UNIT}$$

Figure 1: The monadic intersection type system. By $\mathsf{supp}(e)$ we indicate the subset of A from which $e \in T(A)$ is built, and by $g_{\mathsf{op}} : T^nA \to TA$ the algebraic operation corresponding to the syntactic operation op.

Theorem 1. *Let t be a λ-term. Then $t\Downarrow$ if and only if $\vdash t : M$. Moreover* $\mathsf{obs}(t) = \mathsf{obs}(M)$.

Here, $\mathsf{obs}(\cdot) : T(A) \to T(1)$ is the function that returns the observable behavior of a monadic object (extended to terms as $\mathsf{obs}(t) := \mathsf{obs}(e)$, when $t \Downarrow e \in T(\mathbb{V})$). Indeed, we obtain such a result by generalizing standard soundness and completeness theorems, via abstract relational techniques, allowing for the lifting of subject reduction, expansion, and the reducibility argument.

Some interesting notions of observation, such as the probability of convergence in probabilistic calculi, are naturally *infinitary.* For this reason, we extend our type system to capture infinitary behaviors. Interestingly, we need to add just one rule to the previous (finitary) system, namely:

$$\frac{}{\Gamma \vdash t : \bot}\ \text{BOT}$$

i.e. the one that can type every term with the bottom of the underlying monad (the latter has now to satisfy some domain theoretic properties, such as being dcppo-enriched). Then, a term can be typed in many ways and the characterization of the effectful behavior is obtained as the limit of the approximations.

Theorem 2. *Let t be a λ-term. Then* $\mathsf{obs}(t) = \bigsqcup\{\mathsf{obs}(M) \mid \vdash t : M\}$

Limits. Algebraic effects cover many interesting computational effects, such as nondeterminism, probability, state, and exception throwing. However, our approach is intrinsically limited to a class of well-behaving monads called *weakly cartesian* (morally, those for which there is no loss of information when the bind is applied). Moreover, if one sticks with the finitary case, where the natural notion of convergence is termination, operations that erase arguments are not allowed, since they break the completeness of the system. Still, this restriction can be removed considering the infinitary semantics.

Future Work. At least two interesting (and non trivial) *perspectives* are opened by this work. The former is the application of our system to higher-order model checking [KO09, Kob09] of effectful programs. The latter is the idea of measuring the precise cost of the evaluation of typed terms through their type derivation. This requires switching to the non-idempotent setting [dC18, AGK20], and considering monadic costs (*e.g.* the *expected* cost in the probabilistic setting, or the *maximum* cost in must nondeterminism).

References

[AGK20] Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds, fully developed. *J. Funct. Program.*, 30:e14, 2020.

[BDL18] Flavien Breuvart and Ugo Dal Lago. On intersection types and probabilistic lambda calculi. In David Sabel and Peter Thiemann, editors, *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*, pages 8:1–8:13. ACM, 2018.

[CD78] Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type assignment for λ-terms. *Arch. Math. Log.*, 19(1):139–156, 1978.

[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.

[DCRDR07] M. Dezani-Ciancaglini and S. Ronchi Della Rocca. Intersection and Reference Types. In *Reflections on Type Theory, Lambda Calculus, and the Mind*, pages 77–86. Radboud University Nijmegen, 2007.

[DLFRDR21] Ugo Dal Lago, Claudia Faggian, and Simona Ronchi Della Rocca. Intersection types and (positive) almost-sure termination. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021.

[dLT21a] U. de' Liguoro and R. Treglia. Intersection types for a λ-calculus with global store. In Niccolò Veltri, Nick Benton, and Silvia Ghilezan, editors, *PPDP 2021: 23rd International Symposium on Principles and Practice of Declarative Programming, Tallinn, Estonia, September 6-8, 2021*, pages 5:1–5:11. ACM, 2021.

[dLT21b] Ugo de' Liguoro and Riccardo Treglia. From semantics to types: the case of the imperative lambda-calculus. 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 168–183. Open Publishing Association, 2021.

[DP00] R. Davies and F. Pfenning. Intersection types and computational effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 198–208. ACM, 2000.

[GF21] Francesco Gavazzo and Claudia Faggian. A relational theory of monadic rewriting systems, part I. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, pages 1–14. IEEE, 2021.

[KO09] Naoki Kobayashi and C.-H. Luke Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pages 179–188. IEEE Computer Society, 2009.

[Kob09] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 416–428. ACM, 2009.

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

[PP01] Gordon D. Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.

14

Session 18: Meta-theoretic studies of type systems

The Rewster: The Coq Proof Assistant with Rewrite Rules

Gaëtan Gilbert[1], Yann Leray[1,2], Nicolas Tabareau[1], and Théo Winterhalter[1]

[1] Inria, France
[2] École Normale Supérieure, France

Dependently typed languages such as Coq or Agda are very convenient tools to program with strong invariants and develop mathematical proofs. However, a user might be inconvenienced by things such as the fact that `n` and `n+0` are not considered *definitionally* equal, or the inability to postulate one's own constructs with computation rules such as exceptions [PT18]. Coq modulo theory [Str10] solves the first of the two problems by extending Coq's conversion with decision procedures, *e.g.*, for linear integer arithmetic. Rewrite rules can be used to deal with directed equalities for natural numbers, but also to implement exceptions that compute. They were introduced in Agda [CA16] a few years ago, and later extended to provide more guarantees with a modular confluence checker [CTW19, CTW21].

We present a work-in-progress extension[1] of Coq which supports user-defined rewrite rules. While we mostly follow in the footsteps of the Agda implementation, we also have to face new issues due to the differences in the implementation and meta-theory of Coq and Agda. The most prominent one being the different treatment of universes as Coq supports cumulativity but no first-class universe levels. We will take advantage of this talk to expose our ideas on how to solve the different issues that arise when adding user-defined rewrite rules to a proof assistant by integrating[2] rewrite rules in MetaCoq [SAB+20, SBF+20], building on previous work [CTW19, CTW21].

Rewrite Rules in Coq. We only support rewrite rules whose head symbol is declared as such with the `Symbol` command, which essentially declares an axiom for which we can postulate computation rules. Rules are then declared using the `Rewrite Rule` command. For instance:

```
Symbol pplus : ℕ → ℕ → ℕ.
Rewrite Rule [ n ] ⊢ pplus 0 n ⇒ n.
Rewrite Rule [ n ] ⊢ pplus n 0 ⇒ n.
Rewrite Rule [ n m ] ⊢ pplus (S n) m ⇒ S (pplus n m).
Rewrite Rule [ n m ] ⊢ pplus n (S m) ⇒ S (pplus n m).
```

will declare the parallel plus that computes on both its arguments. On the left of the turnstile (⊢) variables are quantified and can furthermore be annotated with their type. We will illustrate the features and specificities of our implementation below, while exposing the challenges that come with them.

Non-Linearity. For now, we restrict our rewrite rules to be left-linear: each variable can only appear once on the left-hand side of the rule. This appears like a very strong limitation as certain rules are only well typed in presence of non-linearity: *e.g.*, the computation rule for the J eliminator. This can be circumvented by *forcing* certain variables to be equal to an expression:

```
Rewrite Rule [ A u P t (v := u) (B := A) (w := u) ] ⊢ J A u P t v (eq_refl B w) ⇒ t.
```

[1]Available at `https://github.com/yannl35133/coq/tree/rewrite-rules-TYPES`, examples can be found in the `test-suite/success/rewrule.v` file.

[2]Available at `https://github.com/yannl35133/metacoq/tree/rewrite-rules-TYPES`

These equalities are only used to help elaboration figure out implicits in the right-hand side of the rule. They bear no meaning on the rewrite rule itself. For J, this is not a problem, because all well-typed instances of the left-hand side will actually verify these identities, but it is not a guarantee in general. Take for instance the following symbol and rule:

```
Symbol f : ∀ b, if b then ℕ else 𝔹.                Rewrite Rule [ b := true ] ⊢ f b ⇒ 23.
```

The rule will even trigger for the term `f false` meaning that we have something of type $\mathbb{B}$ which reduces to 23, a violation of subject reduction: the rule is simply not type preserving. We will now see that with universes we can run into more subtle problems with respect to type preservation.

Universes and Cumulativity. As exposed before, one major difference between Coq and Agda is the treatment of universes. In Coq, cumulativity means that sharing a common type for the left-hand side and the right-hand side (*i.e.*, the requirement to postulate their propositional equality) is not sufficient to ensure type preservation, even when the rules are confluent.[3] A counter-example would be the following :

```
Symbol id : Type@{v} → Type@{u}.
Rewrite Rule [ ] ⊢ id Type ⇒ Type@{u}.
```

Both sides share the super type `Type@{1 + max u v}`, but no constraints are inferred from the definition of the rule: it works for any level `v` and `u`. In particular it can be used to map terms of type `Type@{u+1}` to `Type@{u}`. This allows the user to create a term `U : U` when they might have thought assuming `Type → Type` was harmless. Not only does this break subject reduction, but also consistency. This raises two challenges to overcome: (1) theoretically we have to come up with a modular criterion to ensure type preservation of the typing rules; (2) in practice we need to be able to collect universe constraints not only on the symbol declaration but also in the rewrite rules themselves.

Reduction Strategies. Coq also features several reduction strategies such as call-by-value (`cbv`), call-by-name (`cbn`) or call-by-need (`lazy`), which are furthermore highly parametrisable (*e.g.*, they can unfold constants or not). Naively grafting a function that matches left-hand sides of rules on top of one of these reduction machines will often lead to incompleteness issues: (1) one has to ensure that subterms (function arguments) are in weak-head-normal form before matching them, which is not the case by default with `cbn` or `lazy`; (2) a function argument brought to normal form, when substituted into another term, may create a deep redex if the pattern is itself deep, which may cause problems with `cbv`. This means that we need to be careful when adding rewrite rules to those, otherwise users may face the frustration of having to type `cbn` several times to fully evaluate redices in a term.

In fact, we believe that proving such properties about reduction strategies would be nice additions to the MetaCoq project. One final challenge we have to face when dealing with rewrite rules in MetaCoq is that we do not know a priori that the rules do not break strong normalisation, while the current implementation and verification in MetaCoq relies on this assumption [SBF⁺20]. We plan to solve this problem simply by no longer relying on this assumption. Instead, we believe that we can first define the reduction machine and the type checker as partial functions so that we may prove correctness on all terminating inputs, using ideas similar to the Braga method [LWM21].

[3] When the system has uniqueness of type, like Agda, confluence is sufficient.

References

[CA16] Jesper Cockx and Andreas Abel. Sprinkles of extensionality for your vanilla type theory. In *Types for Proofs and Programs*, TYPES, 2016.

[CTW19] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. How to tame your rewrite rules. In *Types for Proofs and Programs*, TYPES, 2019.

[CTW21] 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*, 2021.

[LWM21] Dominique Larchey-Wendling and Jean-François Monin. The Braga Method: Extracting Certified Algorithms from Complex Recursive Schemes in Coq. In *Proof and Computation II*, pages 305–386. WORLD SCIENTIFIC, August 2021.

[PT18] Pierre-Marie Pédrot and Nicolas Tabareau. Failure is Not an Option An Exceptional Type Theory. In *ESOP 2018 - 27th European Symposium on Programming*, volume 10801 of *LNCS*, pages 245–271, Thessaloniki, Greece, April 2018. Springer.

[SAB+20] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. To appear in Journal of Automated Reasoning, 2020.

[SBF+20] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *PACMPL*, 4(POPL), 2020.

[Str10] Pierre-Yves Strub. Coq modulo theory. In *Computer Science Logic: 24th International Workshop, CSL 2010, 19th Annual Conference of the EACSL, Brno, Czech Republic, August 23-27, 2010. Proceedings 24*, pages 529–543. Springer, 2010.

On the Metatheory of Subtype Universes

Felix Bradley and Zhaohui Luo

Royal Holloway, University of London, Egham, U.K.
Felix.Bradley@rhul.ac.uk
zhaohui.luo@hotmail.co.uk

Introduction. Power types were first introduced in a seminal paper by Cardelli and formulated the notion of a 'type of subtypes', analogous to that of a power set in set theory. [2] This not only explicitly mechanised bounded quantification, a form of polymorphism wherein the type quantifier is restricted to only subtypes of a given type, but it also served as a method for both the type theory to talk about its own subtyping, and for the subtyping relation to be completed subsumed by the typing relation (by considering $A \leq B$ as a shorthand for $A : \text{Power}(B)$. The type theory in Cardelli's work promoted more powerful expression over more well-behaved metatheoretical properties, such as including **Type** : **Type**, which induces logical inconsistency and non-normalisation [4, 5].

Subtype universes were initially introduced by Maclean and Luo as a way of formalising the notion of a 'type of subtypes' for a well-studied logical type theory equipped with coercive subtyping [9]. This extended type theory has several nice metatheoretical properties such as strong normalisation, but this implementation excluded certain kinds of subtyping relations from being used, and the formulation was wrapped up in the complexities of the underlying type theory's universe hierarchy. In particular, the implementation excludes (in layman's terms) subtyping relations where there are any occurrences of $\mathcal{U}$, or wherein the supertype inhabits a type universe of a smaller level than that of the subtype.

We consider a simpler yet more expressive reformulation of subtype universes by amending a logical type theory equipped with coercive subtyping with the following rules

$$\frac{\Gamma \vdash B\,\text{type}}{\Gamma \vdash \mathcal{U}(B)\,\text{type}} \qquad \frac{\Gamma \vdash A \leq_c B}{\Gamma \vdash \langle A, c\rangle : \mathcal{U}(B)}$$

and two operators σ_1 and σ_2 which respectively extract the type and coercion of a subtype universe's 'pair'.

Expressive Subtype Universes. Prior work on subtype universes included the restriction that a subtype must inhabit a 'smaller' type universe than that of the supertype, and likewise must not include subtype universes. This was necessary in Maclean and Luo's work on extending UTT[$\mathcal{C}$] with subtype universes as their proof of logical consistency and strong normalisation was via an embedding of their extended type theory back into UTT[$\mathcal{C}$], primarily due to the pre-existing hierachy of type universes through which UTT controls its predicative type structure.

This is not necessary in general, however. By considering extending a theory which only possesses an impredicative type universe of propositions Prop, such a type theory has the ability to use higher-level types on either side of the subtyping relation, such as $\mathcal{U}(A) \leq_c A$. Combined with the new operator that extracts a given coercion from a subtyping relation, these new features significantly expand on expressiveness of the type theory, allowing for more subtyping relations than Maclean and Luo's previous implementation.

There still remains the difficulty of proving that a collection of subtyping judgements and inference rules are coherent - "that every possible derivation of a statement $\Gamma \vdash a : A$ has the same meaning." [10] The difficulty of this task is increased for subtype universes when we consider subtyping judgements that include subtype universes, as the coherency of these judgements may now be dependent on other subtyping judgements.

Applications. Subtype universes, in a similar vein to power types, can be used to explicitly mechanise bounded quantification by considering $\lambda(X \leq A).M$ as syntactic sugar for $\lambda(x : \mathcal{U}(A)).[\sigma_1(x)/X]M$, and as such have a variety of uses in modelling programming languages. Additionally, the ability to extract coercions allows for more complex subtyping judgements and inference rules, allowing for a more rich type theory. This has been particularly insightful in formalisation of mathematics, wherein subtype universes have been rudimentarily useful in modelling the topologies of various spaces.

In particular, these additions have proven useful in natural language semantics, wherein types can be interpreted as common nouns and terms as specific instances of these nouns. Here, subtype universes can be used to model subsective adjectives. If CN is the universe of common nouns, then where previously we may have had a term $\text{skillful} : \Pi(A : \text{CN}).A \to \text{Prop}$, we may wish to exclude instances such as skillful(chair). By instead considering $\text{skillful} : \Pi(x : \mathcal{U}(\text{Human})).\sigma_1(x) \to \text{Prop}$ instead, we preserve desired terms such as $\text{skillful}(\langle \text{doctor}, c \rangle)$, and exclude undesired terms such as $\text{skillful}(\langle \text{chair}, c' \rangle)$.

Metatheory. We consider a subset τ of $\text{UTT}[\mathcal{C}]$ with some basic types such as $\mathbf{0}$, $\mathbf{1}$ and $\mathbb{N}$, dependent pair types and dependent function types, and no type universes other than Prop.. We have shown that extending τ with any collection of coherent subtyping judgements $\mathcal{C}$ preserves any underlying strong normalisation and logical consistency. This was done by considering an embedding of $\tau[\mathcal{C}]$ into its parent system $\text{UTT}[\mathcal{C}]$, which has been proven to be strongly normalising and logically consistent [7, 8]. The key idea is that subtype universes $\mathcal{U}(B)$ in our type theory should correspond to types of the form $\Sigma(X : \text{Type}_i).X \to B$ in $\text{UTT}[\mathcal{C}]$.

Describing this embedding is relatively easy when the subtype in a subtyping rule includes fewer uses of $\mathcal{U}$ than the supertype. We call subtyping judgements which satisfy this property 'monotonic', and show that a type theory with subtype universes extended only by coherent and monotonic $\mathcal{C}$ can trivially be embedded into $\text{UTT}[\mathcal{C}]$. For a subset of non-monotonic subtyping rules, we can alter our embedding - if the difference in the number of appearances of $\mathcal{U}$ on either side of a subtyping rule is always bounded by some k, then we can instead 'shift' our embedding by sending $\mathcal{U}(B)$ to $\Sigma(X : \text{Type}_{i+k}).X \to B$ instead.

Conclusion. Our work on generalising and extending subtype universes has allowed for a greater variety of subtyping relations while preserving strong normalisation and logical consistency. There are some open questions we wish to explore further: for example, including a universal supertype **Top** often presents metatheoretic issues, [3] [10] especially in conjunction with coercive subtyping, but it may be possible to develop structured or predicative alternatives which do not.

Our work has also explored models of point-set topology using subtype universes, interpreting $\mathcal{U}(B)$ as the topology of the space B: we would like to further refine this idea, and analyse the role that coercive subtyping plays in this. Finally, Aspinall's work on λ_{Power}, a predicative type theory using power types, has proven fruitful with results such as strong normalisation but has run into difficulties with certain metatheoretical proofs [1]. Hutchins' work on pure subtype systems generalises several subtyping to subsume typing entirely and has proven a very powerful system impredicative system without strong normalisation but with similar difficulties in the metatheory [6]. A better understanding of the complexities of these systems will be key going forward.

References

[1] David Aspinall. Subtyping with power types. In Peter G. Clote and Helmut Schwichtenberg, editors, *Computer Science Logic*, pages 156–171, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[2] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 70–79, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/73560.73566.

[3] Adriana Compagnoni. Higher-order subtyping and its decidability. *Information and Computation*, 191(1):41–103, 2004. URL: https://www.sciencedirect.com/science/article/pii/S0890540104000094, doi:https://doi.org/10.1016/j.ic.2004.01.001.

[4] Thierry Coquand. An analysis of Girard's paradox. Technical Report RR-0531, INRIA, May 1986. URL: https://hal.inria.fr/inria-00076023.

[5] James G. Hook and Douglas J. Howe. Impredicative strong existential equivalent to type : Type. Technical report, Cornell University, Ithaca, New York, USA, June 1986.

[6] DeLesley S. Hutchins. Pure subtype systems. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, page 287–298, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1706299.1706334.

[7] Zhaohui Luo. Coercive subtyping in type theory. In Dirk van Dalen and Marc Bezem, editors, *Computer Science Logic*, pages 275–296, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[8] Zhaohui Luo, Sergei Soloviev, and Tao Xue. Coercive subtyping: Theory and implementation. *Information and Computation*, 223:18–42, 2013. URL: https://www.sciencedirect.com/science/article/pii/S0890540112001757, doi:10.1016/j.ic.2012.10.020.

[9] Harry Maclean and Zhaohui Luo. Subtype Universes. In Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch, editors, *26th International Conference on Types for Proofs and Programs (TYPES 2020)*, volume 188 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/opus/volltexte/2021/13888, doi:10.4230/LIPIcs.TYPES.2020.9.

[10] Benjamin C. Pierce. Bounded quantification is undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, page 305–315, New York, NY, USA, 1992. Association for Computing Machinery. doi:10.1145/143165.143228.

Nominal techniques as an Agda library

Murdoch J. Gabbay[1] and Orestis Melkonian[2]

[1] Heriot-Watt University, Scotland
[2] University of Edinburgh, Scotland

Introduction Nominal techniques [8] provide a mathematically principled approach to dealing with names and variable binding in programming languages. However, integrating these ideas in a practical and widespread toolchain has been slow, and we perceive a chicken-and-egg problem: there are no users for nominal techniques, because nobody has implemented them, and nobody implements them because there are no users. This is a pity, but it leaves a positive opportunity to set up a virtuous circle of broader understanding, adoption, and application of this beautiful technology.

This paper explores an attempt to make nominal techniques accessible as a library in the Agda proof assistant and programming language [9], which can be viewed as a port of the first author's Haskell `nom` package [6], although that would be an injustice as its purpose is two-fold:

1. provide a convenient library to use nominal techniques in Your Own Agda Formalisation
2. study the meta-theory of nominal techniques in a rigorous and *constructive* way

A solution to Goal 1 must be ergonomic, meaning that a *technical* victory of implementing nominal ideas is not enough; we further require a *moral* victory that the overhead be acceptable for practical systems. Apart from this being a literate Agda file, our results have been mechanised and are publicly accessible: `https://omelkonian.github.io/nominal-agda/`.

Nominal setup We conduct our development under some abstract type of **atoms**, satisfying certain constraints, namely decidable equality and being infinitely enumerable.[1]. We model this in Agda using *module parameters*, which could be instantiated with a concrete type:

```
  module _ (Atom : Type) {{ _ : DecEq Atom }} {{ _ : Enumerable∞ Atom }} where
    И : (Atom → Type) → Type
    И φ = ∃ λ (xs : List Atom) → (∀ y → y ∉ xs → φ y)
```

The И **quantifier** enforces that a predicate holds for all but finitely many atoms, and **swapping** of two atoms can be performed on any type, subject to some laws:

```
record Swap (A : Type) : Type where
  field swap : Atom → Atom → A → A
  ⦇_↔_⦈_ = swap
```

```
instance
  ↔Atom : Swap Atom
  ↔Atom .swap x y z =
    if z == x then y else if z == y then x else z
```

```
    record SwapLaws : Type where
      field swap-id   : ⦇ 𝕒 ↔ 𝕒 ⦈ x ≡ x
            swap-rev  : ⦇ 𝕒 ↔ 𝕓 ⦈ x ≡ ⦇ 𝕓 ↔ 𝕒 ⦈ x
            swap-sym  : ⦇ 𝕒 ↔ 𝕓 ⦈ ⦇ 𝕓 ↔ 𝕒 ⦈ x ≡ x
            swap-swap : ⦇ 𝕒 ↔ 𝕓 ⦈ ⦇ 𝕔 ↔ 𝕕 ⦈ x ≡ ⦇ ⦇ 𝕒 ↔ 𝕓 ⦈ 𝕔 ↔ ⦇ 𝕒 ↔ 𝕓 ⦈ 𝕕 ⦈ ⦇ 𝕒 ↔ 𝕓 ⦈ x
```

We only need to provide instances for the base case of *atoms* (whence the decidable equality), and *abstractions* (coming up next). From this we can systematically derive swapping definitions for all user-defined types, using a compile-time macro/tactic (c.f. the case study later on).

One particularly useful family of axioms in equivariant ZFA foundations [5] is that swapping distributes everywhere (constructors, functions, type formers) with the special case for swapping itself being swap-swap. It is consistent to axiomatize this generalized notion of distributivity for swap and we do so by means of a tactic that realises this *axiom scheme*. Most of the time

[1]...also known as "unfiniteness" in a recent nominal mechanization of the locally nameless approach [10].

we are working with types that have **finite support**, expressed using the 'new' quantifier: И² λ 𝕒 𝕓 → swap 𝕓 𝕒 *x* ≡ *x*. We can then define **equivariant** elements that admit the empty support, as well as an operation to generate fresh atoms freshAtom : *A* → *Atom* (whence the module requirement that atoms are infinitely enumerable). Agda is constructive, so freshAtom is constructive too, which is different from how fresh atoms are used in (non-constructive) set theories. An **abstraction** is just a pair of an atom and an element:

```
Abs A = Atom × A

conc : Abs A → Atom → A
conc (𝕒 , x) 𝕓 = swap 𝕓 𝕒 x
```

```
instance
  ↔Abs : Swap (Abs A)
  ↔Abs .swap 𝕒 𝕓 (𝕔 , x) = (swap 𝕒 𝕓 𝕔 , swap 𝕒 𝕓 x)
```

Note that we can also provide a *correct-by-construction* and *total* concretion function. In nominal techniques based on Fraenkel-Mostowski set theory [8] this is impossible, and it seems to be a novel observation that in a constructive setup a total concretion function is fine.

Case study Once equipped with all expected nominal facilities, in particular *atoms* and *atom abstractions*, it is easy to define terms in **untyped λ-calculus** without mentioning de Bruijn indices or anything of that sort. For the sake of ergonomics and efficient theorem proving, we provide a meta-programming macro — based on *elaborator reflection* [2] — that is able to automatically derive the implementation of swapping of any type based on its structure.

```
data Term : Type where
  `_    : Atom → Term
  _·_  : Term → Term → Term
  λ_    : Abs Term → Term
unquoteDecl ↔Term =
  DERIVE Swap [ quote Term , ↔Term ]
```

```
data _≈_ : Term → Term → Type where
  ν≈ : ` x ≈ ` x
  ξ≈ : L ≈ L' → M ≈ M' → L · M ≈ L' · M'
  ζ≈ : И (λ 𝕩 → conc f 𝕩 ≈ conc g 𝕩) → λ f ≈ λ g
```

We can naturally express α-equivalence of λ-terms using the И quantifier and manually prove the aforementioned swapping laws and the fact that every λ-term has finite support. However, these all admit a systematic datatype-generic construction and we are currently in the process of automating them. The rest of the development remains identical to the mechanization presented in the PLFA textbook [14], particularly the 'Untyped' chapter. Meanwhile, the gnarly 'Substitution' appendix involving tedious index manipulations is now replaced by the usual nominal presentation of substitution, alongside a few general lemmas about equivariance and support:

```
_[_:=_] : Term → Atom → Term → Term
(` x)     [ 𝕒 := N ] = if x == 𝕒 then N else ` x
(L · M) [ 𝕒 := N ] = L [ 𝕒 := N ] · M [ 𝕒 := N ]
(λ f)     [ 𝕒 := N ] = λ z ⇒ conc f z [ 𝕒 := N ] where z = freshAtom (𝕒 ∷ supp f ++ supp N)
```

We still have a few remaining lemmas to prove to fully cover the PLFA chapter on untyped λ-calculus, but we do not see any inherent obstacles to completing the confluence proof. A good next step would be to formalise a proof of *cut elimination* for first-order logic, since this involves name-abstraction on both terms and proof-trees.

Related work There have been previous nominal mechanizations in Agda that focus on the concrete instance of the untyped λ-calculus and include a proof of confluence [4, 3]. Ours closely matches the non-mechanized formulation in [7], which the Haskell `nom` package [6] then implements. Another representation of nominal sets in Agda [1] is preliminary and we would hope that our approach is more ergonomic and more amenable to scaling up. We treat our Agda library as a complement to other nominal implementations (in FreshML [12], Isabelle/HOL [13], and Nuprl [11]) that is ergonomic, lightweight, accessible, and illustrates the practical compatibility of nominal techniques within a constructive type system.

References

[1] Pritam Choudhury. Constructive representation of nominal sets in Agda. Master's thesis, Robinson College, University of Cambridge, 2015.

[2] David R. Christiansen and Edwin C. Brady. Elaborator reflection: extending Idris in Idris. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 284–297. ACM, 2016.

[3] Ernesto Copello, Nora Szasz, and Álvaro Tasistro. Machine-checked proof of the Church-Rosser theorem for the lambda calculus using the Barendregt variable convention in constructive type theory. In Sandra Alves and Renata Wasserman, editors, *12th Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2017, Brasília, Brazil, September 23-24, 2017*, volume 338 of *Electronic Notes in Theoretical Computer Science*, pages 79–95. Elsevier, 2017.

[4] Ernesto Copello, Alvaro Tasistro, Nora Szasz, Ana Bove, and Maribel Fernández. Alpha-structural induction and recursion for the lambda calculus in constructive type theory. In Mario R. F. Benevides and René Thiemann, editors, *Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2015, Natal, Brazil, August 31 - September 1, 2015*, volume 323 of *Electronic Notes in Theoretical Computer Science*, pages 109–124. Elsevier, 2015.

[5] Murdoch J. Gabbay. Equivariant ZFA and the foundations of nominal techniques. *J. Log. Comput.*, 30(2):525–548, 2020.

[6] Murdoch J. Gabbay. The `nom` haskell package, 2020. URL: https://hackage.haskell.org/package/nom.

[7] Murdoch J. Gabbay and Aad Mathijssen. A nominal axiomatization of the lambda calculus. *Journal of Logic and Computation*, 20(2):501–531, 2010.

[8] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects Comput.*, 13(3-5):341–363, 2002.

[9] Ulf Norell. Dependently typed programming in Agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009.

[10] Andrew M. Pitts. Locally nameless sets. *Proc. ACM Program. Lang.*, 7(POPL):488–514, 2023.

[11] Vincent Rahli and Mark Bickford. A nominal exploration of intuitionism. In Jeremy Avigad and Adam Chlipala, editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 130–141. ACM, 2016.

[12] Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: programming with binders made simple. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 263–274. ACM, 2003.

[13] Christian Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reason.*, 40(4):327–356, 2008.

[14] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda.* August 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.

15

Session 19: Applications of type theory

Program logics for ledgers

Orestis Melkonian[1,2]*, Wouter Swierstra[3], and James Chapman[2]

[1] University of Edinburgh, Scotland
[2] Input Output, Global
[3] Utrecht University, The Netherlands

Introduction Distributed ledgers nowadays manage substantial monetary funds in the form of cryptocurrencies such as Bitcoin, Ethereum, and Cardano. For such ledgers to be safe, operations that add new entries must be cryptographically sound—but it is less clear how to reason effectively about such ever-growing linear data structures.

We view distributed ledgers as *computer programs*, that, when executed, transfer funds between various parties. As a result, familiar program logics, such as Hoare logic and separation logic, can be defined in this novel setting. Borrowing ideas from concurrent separation logic, this enables modular reasoning principles over arbitrary fragments of any ledger. Our results have been mechanised in the Agda proof assistant [3] and are publicly available:

`https://omelkonian.github.io/hoare-ledgers`

A simple linear ledger We start by studying the simplest form of ledger; assuming an abstract set of participants $\mathcal{P}$, programs are linear sequences of transactions and states S keeps track of everyone's account balance in a finite map.

$$\begin{aligned} T &:= \mathcal{P} \xrightarrow{n} \mathcal{P} \\ L &:= \epsilon \mid T; L \\ S &:= \mathcal{P} \mapsto \mathbb{Z} \end{aligned}$$

```
Alice pays Bob 5;
Alice pays Carroll 10;
Dana pays Alice 2;
...
```

It's straightforward to define denotational, operational, and axiomatic semantics, as well as prove them equivalent to one another. Transactions and ledgers take denotations in the same domain, namely state transition functions $d(t) : S \to S$. For the sake of brevity, we refer to the Agda development for the full definitions and only present the essential Hoare rules:

$$\frac{}{\{P\}\ \epsilon\ \{P\}}\ \textsc{stop} \qquad \frac{\{P\}\ l\ \{Q\}}{\{P \circ d(t)\}\ t; l\ \{Q\}}\ \textsc{step}$$

$$\frac{\{P\}\ l_1\ \{Q\} \qquad \{Q\}\ l_2\ \{R\}}{\{P\}\ l_1 +\!\!+\ l_2\ \{R\}}\ \textsc{app} \qquad \frac{}{\{p_1 \mapsto n\}\ p_1 \xrightarrow{n} p_2\ \{p_2 \mapsto n\}}\ \textsc{send}$$

These should remind you of the corresponding rules for assignment and sequencing in imperative programs from traditional Hoare logic. It is natural to form chains of these Hoare-style state predicates like so: $\{\lambda\sigma.\ \sigma(A) = 2\}\ A \xrightarrow{1} B\ \{\lambda\sigma.\ \sigma(A) = 1\}\ A \xrightarrow{1} C\ \{\lambda\sigma.\ \sigma(A) = 0\}$. However, each step operates on the whole state which is not modular and would not scale to ever-growing blockchain ledgers.

Towards separation We can remedy this by exploiting the monoidal structure of the state space (i.e. pointwise addition of maps $\oplus$), leading to the following notion of *separating conjunction* [4] and the usual Hoare rules of (concurrent) separation logic:

$$(P * Q)(\sigma) := \exists\sigma_1.\ \exists\sigma_2.\ P(\sigma_1)\ \wedge\ Q(\sigma_2)\ \wedge\ \sigma = \sigma_1 \oplus \sigma_2$$

$$\frac{\{P\}\ l\ \{Q\}}{\{P * R\}\ l\ \{Q * R\}}\ \textsc{frame} \qquad \frac{\{P_1\}\ l_1\ \{Q_1\} \qquad \{P_2\}\ l_2\ \{Q_2\}}{\{P_1 * P_2\}\ l_1 \mid\mid l_2\ \{Q_1 * Q_2\}}\ \textsc{par}$$

*Funded by Input Output (iohk.io) through the Edinburgh Blockchain Technology Lab.

Notice the lack of the usual freshness side-conditions, rendering our logic compositional, i.e. a proof about a large ledger can be obtained from proofs about its individual parts.

Extending to the UTxO case If we now try to extend our technique to previous formalisations of UTxO-based blockchain ledgers [1, 2], we are forced to introduce freshness side-conditions when composing *disjoint* ($\uplus$) sub-ledgers and end up with non-compositional rules:

$$\frac{\{P\}\ l\ \{Q\} \qquad l \mathrel{\#} R}{\{P * R\}\ l\ \{Q * R\}}\ \textsc{frame} \qquad \frac{\{P_1\}\ l_1\ \{Q_1\} \qquad \{P_2\}\ l_2\ \{Q_2\} \qquad l_1 \mathrel{\#} P_2 \qquad l_2 \mathrel{\#} P_1}{\{P_1 * P_2\}\ l_1 \,||\, l_2\ \{Q_1 * Q_2\}}\ \textsc{par}$$

This is due to the fact that, in contrast to the previous *by-value* formulation, UTxO transactions reference previous unspent outputs *by name* (requiring the transaction's hash and an index into its outputs), so we need to explicitly track names which enforces a coarse level of modularity.

Abstract UTxO We propose a novel UTxO model that embraces the initial *value-centric* perspective, where previous unspent outputs are only referred to by value (i.e. the monetary value and the script that locks it). This means we now hold bags rather than maps in our state space, abstracting away from the explicit names and hash references of the concrete UTxO model. By exploiting the monoidal nature induced by pointwise bag addition/inclusion, we regain our compositional Hoare rules free of side-effects, as demonstrated in the example below that contrasts a monolithic proof using FRAME (left) to a modular proof combining two smaller proofs using PAR (right):

$$\begin{array}{lll}
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} & & \\
\quad t_1 & \dashv \textsc{frame}(C \mapsto 0 * D \mapsto 1, \textsc{send}) & \\
\{A \mapsto 0 * B \mapsto 1 * C \mapsto 0 * D \mapsto 1\} & & \approx \\
\{C \mapsto 0 * D \mapsto 1 * A \mapsto 0 * B \mapsto 1\} & & \\
\quad t_2 & \dashv \textsc{frame}(A \mapsto 0 * B \mapsto 1, \textsc{send}) & \\
\{C \mapsto 1 * D \mapsto 0 * A \mapsto 0 * B \mapsto 1\} & & \approx \\
\{A \mapsto 0 * B \mapsto 1 * C \mapsto 1 * D \mapsto 0\} & & \\
\quad t_3 & \dashv \textsc{frame}(C \mapsto 1 * D \mapsto 0, \textsc{send}) & \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 1 * D \mapsto 0\} & & \approx \\
\{C \mapsto 1 * D \mapsto 0 * A \mapsto 1 * B \mapsto 0\} & & \\
\quad t_4 & \dashv \textsc{frame}(A \mapsto 1 * B \mapsto 0, \textsc{send}) & \\
\{C \mapsto 0 * D \mapsto 1 * A \mapsto 1 * B \mapsto 0\} & & \approx \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} & & \square
\end{array}$$

$$\begin{array}{c}
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \\
\boxed{\begin{array}{ll}
\{A \mapsto 1 * B \mapsto 0\} & \\
\quad t_1 & \dashv \textsc{send} \\
\{A \mapsto 0 * B \mapsto 1\} & \\
\quad t_3 & \dashv \textsc{send} \\
\{A \mapsto 1 * B \mapsto 0\} & \square
\end{array}}
\;
\boxed{\begin{array}{ll}
\{C \mapsto 0 * D \mapsto 1\} & \\
\quad t_2 & \dashv \textsc{send} \\
\{C \mapsto 1 * D \mapsto 0\} & \\
\quad t_4 & \dashv \textsc{send} \\
\{C \mapsto 0 * D \mapsto 1\} & \square
\end{array}}
\ \dashv \textsc{par} \\
\{A \mapsto 1 * B \mapsto 0 * C \mapsto 0 * D \mapsto 1\} \qquad \square
\end{array}$$

Sound abstraction Even though we believe our abstract UTxO model might be a better foundation for a next-generation blockchain, we still wish to guarantee that it is sound to reason at this higher level in order to prove properties of *concrete* UTxO ledgers that currently exist. To formulate soundness, we start by relating concrete ($\mathbb{C}$) and abstract ($\mathbb{A}$) states, i.e. collect all values of a key-value map in a bag: $abs^S(\sigma) = \{\sigma(k) | k \in \sigma$. A similar construction is defined for ledgers (abs^L). After proving a crucial lemma that connects the concrete and abstract denotational semantics (left), we can finally prove the *soundness theorem* (right):

$$\frac{\mathbb{C}[\![\ l\ ]\!](\sigma) = \tau}{\mathbb{A}[\![\ abs^L(l)\ ]\!](abs^S(\sigma)) = abs^S(\tau)} \qquad \frac{\mathbb{A}\{P\}\ abs^L(l)\ \{Q\} \qquad l \text{ valid in } \sigma}{\mathbb{C}\{P \circ abs^S\}\ l\ \{Q \circ abs^S\}}\ \textsc{soundness}$$

Conclusion The use of a proof assistant was instrumental in navigating various points in the design space; we are now confident that our approach lays robust foundations at the ledger level and is able to culminate into larger-scale verification of actual smart-contracts.

References

[1] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of Bitcoin transactions. In Sarah Meiklejohn and Kazue Sako, editors, *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018, Revised Selected Papers*, volume 10957 of *Lecture Notes in Computer Science*, pages 541–560. Springer, 2018.

[2] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The Extended UTXO model. In Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security - FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers*, volume 12063 of *Lecture Notes in Computer Science*, pages 525–539. Springer, 2020.

[3] Ulf Norell. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

[4] John C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

Formalization of Blockchain Oracles in Coq

Mohammad Shaheer[1], Giselle Reis[1], Bruno Woltzenlogel Paleo[2], and Joachim Zahnentferner[2]*

[1] Carnegie Mellon University, Qatar – mshaheer@andrew.cmu.edu, giselle@cmu.edu
[2] Djed Alliance – bruno.wp@gmail.com

Abstract

Oracles are crucial components that bring external data to smart contracts deployed on blockchains. With the recent surge in popularity of decentralized finance (DeFi) applications, it is critical to provide assurances about the oracle implementations as these applications deal with high-value transactions and a small price discrepancy can lead to huge losses. Although there are many oracle implementations, there have not been many efforts to formally verify their behavior. We present a simple oracle implementation in Solidity and its formal model using the Coq interactive theorem prover. We also prove interesting trace-level properties that give us formal guarantees about the oracle's behavior at a high level. Our work can be a stepping stone for future oracle implementations and provide developers with a framework for formally verifying their implementations.

Smart contracts are programs that run on blockchains, maintain an internal state and provide functions whose execution may depend on the internal state and on calls to functions of other smart contracts. Due to the decentralized operation of blockchains, applications built through smart contracts may enjoy desirable qualities such as transparency, censorship resistance and interoperability. The flexible programmability of smart contracts coupled with the qualities of the blockchain environment formed a fertile ground for financial applications, where such qualities are highly desirable and at times absent in the traditional financial sector [1]. This ushered an era of so-called *Decentralized Finance* (DeFi). Thousands of digital assets have been implemented as ERC20 [2] smart contracts that maintain, as their internal states, the balances of all users of an asset and provide functions for transferring amounts from one user to another. Collateralized lending applications [3] have been developed to allow users to lend and borrow such assets. Exchange applications [4] have been developed to allow users to swap assets. And various stablecoin algorithms have been proposed to allow the price of a digital asset to track the price of another asset.

Blockchain applications often need access to information that is not readily available on the blockchain. For DeFi applications in particular, data from the external world can be crucial. For example: a stablecoin needs to know the relative price of the fiat currency to which it is pegged; a collateralized lending application needs to know the value of the collateral to know when to liquidate debts. This need is satiated through a special type of blockchain application known as *oracle*. An oracle has a smart contract that maintains the desired data in its internal state and implements an *oracle protocol* that establishes the conditions under which various entities may write or read data from the contract. Some of these entities are also responsible for operating off-chain components of the oracle to obtain the data to be written to the contract.

Oracle protocols differ widely in how frequently the data is updated, how data consumers are charged for reading the data, how data being written by multiple sources is aggregated, how misbehaviour is penalized and desirable behaviour incentivized.

Unfortunately, existing oracle implementations are ad hoc, lacking a formal definition, or even a precise description, of their protocol. Without a formal definition, it is harder to provide guarantees for the oracle's functioning. In the worst case, there might even be unidentified exploitable security vulnerabilities. This leads to uncertainty and compromises confidence on the applications that depend on such oracle implementations.

Our work tackles this issue by: (1) proposing a (non-exhaustive) set of desirable properties for oracle protocols; (2) formalizing a simple oracle protocol in Coq and formally proving that it satisfies these desirable properties; (3) implementing an oracle smart contract in Solidity closely following the formal oracle protocol.

Our work focuses on the long-term economic sustainability of the oracle. Since an oracle has off-chain components that are costly to operate (e.g. to obtain data from an external source and write it to the contract), the entities operating these components need to be economically incentivized to keep doing their work. This is done by charging fees from the data consumers who read data from the oracle. The oracle protocol automatically adjusts the fees for reading data based on the costs of the data provider and the frequency of data reads, aiming to ensure that: (A) the costs of the data provider are covered by the fee revenue from data reads; and (B) every data consumer is paying a price that is fair in the sense that they never have to pay twice for the same data point and the cost of data is distributed evenly between all data consumers.

Given the desired properties (A) and (B), we developed an oracle smart contract in Coq in parallel with its Solidity implementation assuming a single external data provider. The Coq formalization was designed such that properties could be proved fairly easily, but also that it faithfully represents the Solidity implementation. Striking this balance is not always straightforward. We went over a few different representations before establishing what follows.

Since Solidity is an object-oriented language, the formalization uses Coq's `Record`s to represent objects. The oracle contract itself is an object, which is implemented in Coq as a `Record` called `State`. This record consists of two sub-records: `OracleState` and `OracleParameters`; and a `Trace` list. `OracleState` contains contract attributes that change with time and `OracleParameters` encompasses immutable attributes set at initialization. `Trace` is a list of `Event`s that keeps a record of the operations performed on the contract. Every time a contract function is called, its corresponding `Event` is added to the `Trace`. Finally, in order to account for side effects, contract functions implemented in Coq have explicit `State`s in their input and output.

Our oracle protocol uses a subscription-based model where consumers deposit credit before reading the data. The cost of a data read is taken from a consumer's balance and it depends, among other things, on a *base fee*, which can be adjusted but may not exceed a *maximum fee*. These fees are part of the contract's parameters and can be adjusted by the data provider. Proper adjustments can, under certain assumptions, provably ensure that properties (A) and (B) are fulfilled.

Using the oracle formalization in Coq, we could prove two main theorems:

Thm 1. *For all consumers c, if* $\mathsf{credit}(c) \geq 0$*, then after any contract function call* $\mathsf{credit}(c) \geq 0$.

Thm 2. *Between two consecutive data writes, each consumer pays once to read the data.*

Both properties are proved using induction on the `Trace`. The proof of Theorem 1 uses two helper lemmas. The proof of Theorem 2 uses nine helper lemmas. Both proofs also use a number of auxiliary definitions for manipulating states and traces.

The implementation and formalization can be found at, respectively, `https://github.com/DjedAlliance/Oracle-Solidity/tree/cmu-qatar` and `https://github.com/DjedAlliance/Oracle-FormalMethods`.

For future work, our next step is to shift our focus from economic aspects to governance aspects around the whitelist of data providers, who can adjust the oracle's parameters and vote to add or remove data providers from the whitelist. We plan to prove theorems related to the security of such governance processes under circumstances where some data providers may have been compromised.

References

[1] Bowen Liu, Pawel Szalachowski, and Jianying Zhou. A First Look into DeFi Oracles, 2020.

[2] ERC20 White Paper. https://erc20.tech/erc20token-whitepaper, Accessed on Mar. 2023.

[3] Robert Leshner and Geoffrey Hayes. Compound: The money market protocol. https://eauapcddxck6wa2w2ufzqnjvg7u7bgw22fh6fkdkwooyeh7twypq.arweave.net/ICgHiGO4lesDVtULmDU1N-nwmtrRT-KoarOdgh_zth8/documents/Compound.Whitepaper.pdf, Accessed on Mar. 2023.

[4] Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core. https://berkeley-defi.github.io/assets/material/Uniswap%20v3%20Core.pdf, Accessed on Mar. 2023.

Notes

*. "Joachim Zahnentferner" is the pseudonym used by Bruno Woltzenlogel Paleo for papers on the topic of blockchains and cryptocurrencies. He continues to use his own name for papers on topics related to logic and formal methods.

A simple model of smart contracts in Agda

Fahad F. Alhabardi[1] and Anton Setzer[2]

[1,2]Swansea University, Dept. of Computer Science
[1]fahadalhabardi@gmail.com
[2]a.g.setzer@swansea.ac.uk   http://www.cs.swan.ac.uk/~csetzer/

Abstract

This paper defines a simple basic model of smart contracts in Agda. In the previous work [1], we verified smart contracts in Bitcoin. The aim of this paper is the first step towards transferring this work to the Solidity-style [8] smart contracts of Ethereum, namely, to develop a model. This model is much more complex than that used for Bitcoin because contracts in Ethereum are object-oriented. We build a simple model which supports simple execution, calling of other contracts and functions and which refers to addresses and messages.

Smart contracts are one of the main applications of Blockchain. In Blockchain, smart contracts are programs that automatically run when certain predetermined criteria are satisfied [9, 5]. The simplest example of a smart contract is used to buy and sell goods or services: The purchaser deposits funds on the Blockchain for the seller. The funds are not released to the seller until a second signature is obtained from the purchaser upon receipt of the items. If the products are not delivered on time, the customer is refunded [7]. Simple smart contracts like the above can be written in Bitcoin in Script [4]. The cryptocurrency Ethereum has a much more powerful Turing complete machine language, the Ethereum Virtual Machine (EVM), which allows calls to other contracts. There are two high-level languages which compile into the EVM, Solidity and Vyper [2, 3]. Other cryptocurrencies have their own languages [2].

As smart contract failures can trigger huge financial losses, accuracy and security are compulsory in smart contracts before deploying on the Blockchain network because once deployed, the contract is immutable. [10]. Verification of smart contracts is costly, yet it is invaluable in helping to limit the financial implications of poorly designed contracts. An example of poor design is evident from hacking the Distributed Autonomous Organization (DAO) smart contract in 2016 [6, 7]. DAO is a contract issued on the cryptocurrency Ethereum and is an investor-directed venture capital fund based on smart contracts. A flaw in the smart contract code of DAO was exploited by cyber criminals when the fund's market value reached US$ 150 million.

In this paper, we build as a first step towards the verification of smart contracts a model of smart contracts in the theorem prover Agda. In the Ethereum virtual machine, one can call functions using arguments which serialise the data passed on to them. In our model, we abstract from this by defining a data type of messages. Messages are natural numbers, or lists of messages. This allows to represent elements of data types as messages; for instance, an array can be represented as a list of messages, and a map can be represented as a list of pairs consisting of a key and the element it is mapped to (both represented as messages).

```
data Msg : Set where nat : (n : ℕ) → Msg
                     list : (l : List Msg) → Msg
```

After that, we define smart contracts mutually recursively as a coinductive record SmartContractExecStep, which allows conditionals, sequential compositions, and loops, together with the data type SmartContractExec as follows:

```
record SmartContractExecStep : Set where
  coinductive
```

```
    field  calledAddress  : Address
           calledFunction : FunctionName
           calledMsg      : Msg
           cont           : Msg → SmartContractExec
  data SmartContractExec : Set where
    return : Msg → SmartContractExec
    call   : SmartContractExecStep → SmartContractExec
    error  : ErrorMsg → SmartContractExec
```

Our smart contracts execution step consists of the address to call (calledAddress), the function name to call (calledFunction), followed by the message call (calledMsg), and the continuation (cont) which determines the next execution step depending on the message returned when the call to the function has finished. SmartContractExec determines the three steps how to continue in the execution: return, which will terminate execution and return its argument, call which will make a call SmartContractExecStep, and then continue as defined by its continuation argument, and error, which will return an error.

The Ledger is a function which depending on addresses, function names, and messages (which are the arguments to the function) returns a SmartContractExecStep:

```
  Ledger = Address → FunctionName → Msg → SmartContractExec
```

In order to compute the execution of a call to a smart contract, we define a smart contract stack (ExecutionStack), each element of which determines depending on the result of the current execution the next SmartContractExecStep:

```
  ExecutionStack = List (Msg → SmartContractExec)
```

The state of executing consists of the execution stack and the current code to be executed:

```
  record StateExecFun : Set where
    constructor stateEF
    field executionStack : ExecutionStack
          nextstep       : SmartContractExec
```

We define stepEF, the one step execution of a smart contract, and stepEFntimes, which iterates it n times, corresponding to execution with a simple form of gas limit:

```
  stepEF       : Ledger → StateExecFun → StateExecFun
  stepEFntimes : Ledger → StateExecFun → ℕ → StateExecFun
```

As an example, we build a ledger which has at address 0 a function "f1" which calls contract with address 1, function "g1", message (nat n) and will terminate with the result returned. Furthermore it has at address 1 a function "g1" which just increments a natural number argument by 1. For all other addresses, functions, and arguments, it is undefined:

```
  testLedger : Ledger
  testLedger 0 "f1" (nat n) = call (smartContractExecStep 1 "g1" (nat n) return)
  testLedger 1 "g1" (nat n) = return (nat (suc n))
  testLedger ow ow' ow"     = error (strErr " Error undefined")
```

To conclude, we have built a basic smart contract model that supports execution. In the next step, we will add state, gas cost and amount of money. Furthermore, we will include more complex operations, such as transactions, in our model and deal with interactive programs in Agda. Moreover, we will verify smart contracts in our model by using the weakest preconditions, extending the work in [1]. Weakest preconditions can be used to determine for a smart contract the conditions required to carry out a certain transfer.

References

[1] Fahad F. Alhabardi, Arnold Beckmann, Bogdan Lazar, and Anton Setzer. Verification of Bitcoin Script in Agda Using Weakest Preconditions for Access Control. In *27th International Conference on Types for Proofs and Programs (TYPES 2021)*, volume 239 of *LIPIcs*, pages 1:1–1:25, Dagstuhl, Germany, 2022. Leibniz-Zentrum für Informatik. doi: https://doi.org/10.4230/LIPIcs.TYPES.2021.1.

[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] Massimo Bartoletti and Roberto Zunino. BitML: A Calculus for Bitcoin Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 83–100, New York, NY, USA, 2018. Association for Computing Machinery. doi:http://dx.doi.org/10.1145/3243734.3243795.

[4] 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.

[5] 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.

[6] Zeinab Nehaï, Pierre-Yves Piriou, and Frédéric Daumas. Model-Checking of Smart Contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 980–987, 2018. doi:http://dx.doi.org/10.1109/Cybermatics_2018.2018.00185.

[7] Anton Setzer. Modelling Bitcoin in Agda. *CoRR*, abs/1804.06398, 2018. URL: http://arxiv.org/abs/1804.06398, arXiv:1804.06398.

[8] Solidity Community. Solidity documentation, Retrieved 10 March 2023. Availabe from https://docs.soliditylang.org/en/v0.8.19/.

[9] 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.

[10] 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.

Rank-polymorphic arrays within dependent types

Artjoms Šinkarovs[1]*and Sven-Bodo Scholz[2]

[1] Heriot-Watt University
`a.sinkarovs@hw.ac.uk`
[2] Radboud University
`svenbodo.scholz@ru.nl`

Many array languages such as APL [6], J [10], Futhark [4], or SaC [9] cater for multi-dimensional arrays as first class citizens. These languages have two main advantages. Firstly, they give rise to concise specifications of numerical algorithms that use array combinators rather than explicit indexing as often found in imperative languages such as Fortran or C. Secondly, as arrays have a very regular structure, many computations on arrays can be automatically parallelised, which leads to efficient executions on a range of parallel platforms [2, 3, 8, 5].

Rank polymorphism is the ability of functions to be applied to arrays of arbitrary ranks. Rank polymorphism is important for two reasons. Firstly, it gives rise to more general array combinators such as map, fold, take, transpose, *etc.* Secondly, the structure of the nesting can be used to enforce non-trivial traversals through sub-arrays which is often the basis for advanced parallel algorithms such as scan or blocked matrix multiply.

In this talk we present how rank-polymorphic arrays can be embedded within a dependently-typed language. On the one hand, our embedding offers the generality of the specifications found in array languages. On the other hand, we guarantee safe indexing and offer a way to reason about concurrency patterns within the given algorithm.

We present the key ingredients of the array framework in Agda. We start with the definition of an array theory.

```
record Array : Set₁ where
  field
    S : Set
    P : S → Set
    ι : ℕ → S
    _⊗_ : S → S → S
    ι-↔ : ∀ {n} → P (ι n) ↔ Fin n
    ⊗-↔ : ∀ {s p} → P (s ⊗ p) ↔ (P s × P p)
    Ar : S → Set → Set
    Ar-↔ : ∀ {s X} → (P s → X) ↔ Ar s X
```

Array shapes S are binary trees with natural numbers as leaves. Array indices P are indexed by shapes, representing trees of natural numbers of the same shape as the index, but where all leaves are component-wise smaller than the shape components. For example, for some shape $(\iota\ a \otimes (\iota\ b \otimes \iota\ c))$ the index is of the form $(i, (j, k))$ where $i < a$, $j < b$ and $k < c$. Array theory does not insist on a particular implementation of S and P but it requires the chosen implementation to be isomorphic to such trees (ι-↔ and ⊗-↔). Finally, arrays (Ar) are indexed by the shape and the element type, and we ask that arrays are representable functors (Ar-↔).

By expanding isomorphisms in the array theory, we get a number of useful array combinators as model constructions. For some $(A : \text{Array})$, we have:

*This work is supported by the Engineering and Physical Sciences Research Council through the grant EP/N028201/1.

```
imap : ∀ {s} → (P s → X) → Ar s X
_[_]   : ∀ {s} → Ar s X → (P s → X)
nest : ∀ {s p} → Ar (s ⊗ p) X → Ar s (Ar p X)
unnest : ∀ {s p} → Ar s (Ar p X) → Ar (s ⊗ p) X
```

By noticing that for a fixed shape s Ar is an applicative functor [7], we obtain operations like:

```
map : ∀ {s} → (X → Y) → Ar s X → Ar s Y
zipWith : ∀ {s} → (X → Y → Z) → Ar s X → Ar s Y → Ar s Z
```

Next, we define an inductive reshape relation that gives rise to reversible permutations of array elements. There can be various reshaping relations allowing more or less liberal permutations.

```
data Reshape : S → S → Set
```

The Reshape relation gives rise to actions on array indices and arrays themselves:

```
_⟨_⟩ : ∀ {s p} → P p → Reshape s p → P s
reshape : ∀ {a b X} → Reshape a b → Ar a X → Ar b X
```

We finish this abstract by presenting the simplest example that demonstrates the power of the proposed framework — a blocked matrix-vector multiplication. We work in the initial model of our array theory, and we assume that we have two functions (`_⊠_ : X → Y → Z`) and (`sum : ∀ {s} → Ar s Z → Z`). Then, a straight-forward matrix-vector multiplication is given by mat-vec-canon. We define the blocked version mat-vec by running induction on s. When s is a singleton we use the canonical multiplication defined earlier, but when s is a product $(s \otimes p)$, we block the matrix into s matrices of p rows and apply *mat-vec* recursively on each block. All these recursive applications can be executed in parallel, as there are no dependencies between the parts.

```
mat-vec-canon : Ar (s ⊗ ι n) X → Ar (ι n) Y → Ar s Z
mat-vec-canon a v = imap λ i → sum $ imap λ k → a [ i ⊗ k ] ⊠ v [ k ]

mat-vec : Ar (s ⊗ ι n) X → Ar (ι n) Y → Ar s Z
mat-vec {s = ι m}     a v = mat-vec-canon a v
mat-vec {s = s ⊗ p} a v = unnest $ map (flip mat-vec v) (nest $ tile a)
```

We can demonstrate that our blocked algorithm computes the same results (using point-wise equality $\_\approx_a\_$); and that the blocked algorithm is stable under reshapes. That is, for all possible reshapes, computing blocked mat-vec on a reshaped array is the same as computing mat-vec on the original array and then performing the reshape.

```
mat-vec-ok : (a : Ar (s ⊗ ι n) X) → (v : Ar (ι n) Y) → mat-vec a v ≈ₐ mat-vec-canon a v
mat-vec-stable : (r : Reshape s p) → (a : Ar (s ⊗ ι n) X) → (v : Ar (ι n) Y)
               → mat-vec (reshape (r ⊕ eq) a) v ≈ₐ reshape r (mat-vec a v)
```

In practice this means, that we can use array reshaping as a vehicle to control which sub-arrays will be executed in parallel. In the talk we will make this idea precise, explaining how exactly one can reason about parallel execution.

We conclude with the observation that the presented array theory is very similar to categories with families [1] which are often used to define type theories. In this analogy, contexts are shapes, substitutions are reshapes, and well-scoped terms are arrays.

References

[1] Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Escardó. On generalized algebraic theories and categories with families. *Math. Struct. Comput. Sci.*, 31(9):1006–1023, 2021.

[2] Clemens Grelck. Shared memory multiprocessor support for functional array processing in sac. *Journal of Functional Programming*, 15(3):353–401, 2005.

[3] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. Breaking the gpu programming barrier with the auto-parallelising sac compiler. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, page 15–24, New York, NY, USA, 2011. Association for Computing Machinery.

[4] Troels Henriksen. *Design and Implementation of the Futhark Programming Language.* PhD thesis, University of Copenhagen, Universitetsparken 5, 2100 København, 11 2017.

[5] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

[6] Kenneth E. Iverson. *A Programming Language.* John Wiley & Sons, Inc., New York, NY, USA, 1962.

[7] CONOR MCBRIDE and ROSS PATERSON. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

[8] Trevor L. McDonell, Manuel M T Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising Purely Functional GPU Programs. In *ICFP '13: The 18th ACM SIGPLAN International Conference on Functional Programming.* ACM, September 2013.

[9] Sven-Bodo Scholz. Single assignment c: Efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, November 2003.

[10] Roger Stokes. Learning j. an introduction to the j programming language. `http://www.jsoftware.com/help/learning/contents.htm`, 2015. [Accessed 2023/02].

16

Session 21: Proof assistants and proof technology

Manifest Termination[*]

Assia Mahboubi[1] and Guillaume Melquiond[2]

[1] Nantes Université, École Centrale Nantes, CNRS, Inria, LS2N, UMR 6004, 44000 Nantes, France
[2] Université Paris Saclay, CNRS, ENS Paris Saclay, Inria, LMF, 91190 Gif-sur-Yvette, France

In formal systems combining dependent types and inductive types, such as the Coq proof assistant [8], non-terminating programs are frowned upon. They can indeed be made to return impossible results, thus endangering the consistency of the system [7], although the transient usage of a non-terminating Y combinator, typically for searching witnesses, is safe [4]. To avoid this issue, the definition of a recursive function is allowed only if one of its arguments is of an inductive type and any recursive call is performed on a syntactically smaller argument. If there is no such argument, the user has to artificially add one, *e.g.*, an accessibility property. Free monads can still be used to address general recursion [6] and elegant methods make possible to *extract* partial functions from sophisticated recursive schemes [2, 5]. The latter yet rely on an inductive characterization of the domain of a function, and of its computational graph, which in turn might require a substantial effort of specification and proof.

This leads to a rather frustrating situation when computations are involved. Indeed, the user first has to formally prove that the function will terminate, then the computation can be performed, and finally a result is obtained (assuming the user waited long enough). But since the computation did terminate, what was the point of proving that it would terminate? This abstract investigates how users of proof assistants based on variants of the Calculus of Inductive Constructions could benefit from manifestly terminating computations. A companion file showcasing the approach in the Coq proof assistant is available on-line [1].

Iteration. Traditional call-by-value programming languages allow a `fix` operator:

```
let rec fix f x = f (fix f) x
```

As this definition is typically forbidden in our setting, we resort to a more domain theoretic approach, so as to enable reasoning about a term y such that $\mathtt{fix}\,F\,x$ terminates on y for a certain x. Consider two types T and U and a function $F : (T \to U) \to (T \to U)$. Given an integer n, a variable $k : T \to U$, and an input $x : T$, the computation of $F^n\,k\,x = (F \circ \ldots \circ F)\,k\,x$ reduces to a value y of type U. If no occurrence of the variable k appears in y, then y is also the result of $F^m\,k\,x$ for any $m \geq n$, which we denote $F^*\,x \rightsquigarrow y$.

Since Coq can compute the normal form of $F^n\,k\,x$ (*i.e.*, `Nat.iter n F k x`) for some concrete n, the property $\forall k,\ F^n\,k\,x = y$ holds by reflexivity. But this equality is only a means to an end. The next step is to prove some properties about y so that it can be used inside some other proof. For a predicate $P : T \to U \to \mathit{Prop}$ that relates inputs and outputs, and from the fact $F^*\,x \rightsquigarrow y$, we can derive $P\,x\,y$ by applying Lemma 1 (whose proof is actually trivial).

Lemma 1. *If* $\exists f, \forall x, P\,x\,(f\,x)$ *and* $\forall k, (\forall x, P\,x\,(k\,x)) \Rightarrow \forall x, P\,x\,(F\,k\,x)$, *then* $\forall n\,x\,y, (\forall k, F^n\,k\,x = y) \Rightarrow P\,x\,y$.

As an illustration, we define an efficient implementation of factorial over binary relative integers (type `Z` in Coq) and, using Lemma 1, we easily prove that, when it terminates on a non-negative input, it does indeed compute its factorial. We then use this to prove a definitional

[*]This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101001995).

identity whose type-checking would take astronomical time. Note that at no point do we need to formally prove that `factZ` terminates over non-negative integers to use it. It just does.

```
Definition factZ k n := if Z.eqb n 0 then 1 else Z.mul n (k (n - 1)).
Lemma factZ_spec n x y : (forall k, Nat.iter n factZ k x = y) → (0 <= x) →
  Z.of_nat ((Z.to_nat x)!) = y. (* n! defines the reference factorial, on type nat *)
Goal Z.of_nat ((Z.to_nat 15)!) = 1307674368000. Proof. exact: (factZ_spec 20). Qed.
```

Accessibility. The previous approach is effective, but it hinges on the fact that one can exhibit a function f such that $\forall x, P\,x\,(f\,x)$. (Above, we used `fun x => Z.of_nat ((Z.to_nat x)!.`) The companion file illustrates how to compute the function McCarthy91 using its recursive definition, but using the non-recursive equivalent definition as f.

In general, the function f is recursive, and thus one has to prove its termination, which might in some cases be just as hard as proving the termination of F^* itself. Those cases thus require a different approach. Consider the following Coq function, inspired from Charguéraud [3]. (See the companion file for the exact but less readable term.)

```
Definition fixacc (dummy : U) F (R : T →T →bool) (x : T) : Acc R x →U :=
  Acc_rect (fun x k ⇒ F (fun y ⇒ if R y x then k y else dummy) x).
```

In addition to the already known arguments F and x, this function takes a Boolean relation R and a proof that x is accessible using this relation (*i.e.*, no infinite decreasing chain from x). This accessibility proof is used to fuel the recursive calls. If at some point on input u, F tries to perform a recursive call with input v, the inner function will check that v is indeed smaller (*i.e.*, $R\,v\,u = \mathit{true}$). If so, it allows the recursive call $k\,v$. Otherwise, it returns a dummy value from U. Note that in practice U will be non-empty, for there is a y such that $F^*\,x \rightsquigarrow y$.

The relation R represents the call graph of the computation $F^*\,x \rightsquigarrow y$. More precisely, $R\,v\,u = \mathit{false}$ if and only if F was invoked on u but it did not perform a direct recursive call with input v. In other words, the above function behaves the same as `Acc_rect (fun u k => F k u)`, which is just F^*. As for the accessibility of x by R, it is a consequence of the fact that the computation actually terminated and thus that the call graph is finite and acyclic. This gives rise to the following consistent axiom, where `fixrel F x y` stands for $F^*\,x \rightsquigarrow y$:

```
Parameter fixrel : forall {T U}, ((T →U) →T →U) →T →U →Prop.
Axiom fixrel_spec : forall {T U} F x y, fixrel F x y →exists R,
  (forall v k1 k2, (forall u, R u v = true →k1 u = k2 u) →F k1 v = F k2 v) /\
  exists W, forall u, y = fixacc u F R x W.
```

Using this axiom, it becomes possible to prove a lemma such as "`fixrel factZ x y -> 0 < y`" without first exhibiting a terminating function or proving termination of `factZ`. Instead, proofs go by unfolding `fixacc`, and by exhibiting two functions k_1 and k_2 that are equal for every input v except when $R\,v\,u = \mathit{false}$. This way, we construct an *ad hoc* proof for every sensible F, *i.e.*, for functionals that do not perform any irrelevant recursive call (*e.g.*, $k\,x - k\,x$).

Conclusion. Checking the antecedent `(fixrel F x y)` of the axiom `fixrel_spec` is easy to do in the kernel of Coq, *e.g.*, in the bytecode interpreter. This could obviously lead to non-terminating computations when checking proofs, but from the user's viewpoint, there is not much difference between computations that cannot terminate and computations that take too long to terminate.

At this point, the open questions are whether there exists a simpler version of this axiom, and whether one can deduce from it a generic variant of Lemma 1.

References

[1] https://fresco.gitlabpages.inria.fr/divers/fix.v.

[2] Ana Bove and Venanzio Capretta. A type of partial recursive functions. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *21st International Conference in Theorem Proving in Higher Order Logics*, volume 5170 of *Lecture Notes in Computer Science*, pages 102–117. Springer, August 2008.

[3] Arthur Charguéraud. Proof pearl: A constructive fixed point combinator for recursive functions, March 2009.

[4] Herman Geuvers, Erik Poll, and Jan Zwanenburg. Safe proof checking in type theory with Y. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 439–452. Springer, 1999.

[5] Dominique Larchey-Wendling and Jean-François Monin. The Braga method: Extracting certified algorithms from complex recursive schemes in Coq. In *Proof and Computation II*, pages 305–386. World Scientific, August 2021.

[6] Conor McBride. Turing-completeness totally free. In Ralf Hinze and Janis Voigtländer, editors, *12th International Conference on Mathematics of Program Construction*, volume 9129 of *Lecture Notes in Computer Science*, pages 257–275. Springer, June 2015.

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

[8] The Coq Development Team. The Coq proof assistant, September 2022.

Extending cAERN to spaces of subsets[*]

Michal Konečný[1], Sewon Park[2], and Holger Thies[2]

[1] Aston University, UK
[2] Kyoto University, Japan

Software packages for exact real computation provide data types to work exactly with real numbers. The AERN library [Kon21], a Haskell library for efficient exact real computation developed and maintained by one of the authors, provides a type `CReal` for real numbers and operations such as arithmetic and limits of fastly converging sequences. For an object of type `CReal`, one can output rational approximations of arbitrary (absolute) precision. Comparisons $x < y$ of two real numbers return an object of type `CKleenean`. A `CKleenean` is interpreted as an element of Kleene's three valued logic over True, False and $\perp$, where $\perp$ stands for the third truth value whose operational meaning is nontermination. In AERN, a `CKleenean` can be evaluated with a certain effort which either returns a Boolean value or undefined.

The cAERN library [KPT22] is a formalization of exact real computation in the Coq proof assistant. (The code can be found on `https://github.com/holgerthies/coq-aern` with parts relevant for the current work contained in the *new-subsets* branch.) One of the main goals of the library is to extract efficient verified Haskell programs built on top of the AERN framework. To achieve this, we axiomatically define e.g. types R of real numbers and K of Kleeneans. We further extend Coq's program extraction to map those types and basic operations on them to corresponding types and operations in AERN. In [KPT22], the soundness of our axiomatization is shown by interpreting our results in a simplified type theory and extending a realizability interpretation in the category of assemblies over Kleene's second algebra.

In recent ongoing work, we extend cAERN to computations on higher objects such as spaces of real functions and hyperspaces of real subsets. To this end, we first express subsets classically as relations $X \to \mathsf{Prop}$ and then assign computational content to such sets via topological notions. Note that we assume classical axioms to hold in Prop.

The Sierpinski space is the two point space $\mathbb{S} := \{\top, \perp\}$ where $\{\top\}$ is the only non-trivial open. Sierpinski space is typically used in computability theory to characterize semi-decidable propositions. In computable analysis, Sierpinski space is used to characterize subsets of represented spaces [Pau16]. For example, a set $A \subseteq X$ is (computably) open if its characteristic function $\chi_A : X \to \mathbb{S}$ is computable.

As Kleeneans K already exists in our system, instead of introducing Sierpinski space as another primitive type, we define it as

$$\mathsf{S} :\equiv \Sigma(b : \mathsf{K}).\ b \neq \mathsf{false}$$

and identify open subsets of a space X with functions $X \to \mathsf{S}$:

$$\mathsf{open}\ A :\equiv \Sigma(f : X \to \mathsf{S}).\ \Pi(x : X).\ f(x) = \top \leftrightarrow x \in A\ .$$

Using basic properties of the Kleeneans, we get short and elegant proofs for simple properties of open sets. However, from the point of view of doing actual computations, using Sierpinski valued functions to represent basic objects is often far from optimal and programs extracted from

[*]Holger Thies is supported by JSPS KAKENHI Grant Numbers 20K19744 and 23H0334. Sewon Park is a JSPS International Research Fellow supported by JSPS KAKENHI (Grant-in-Aid for JSPS Fellows) 22F22071. This project has received funding from the EU's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 731143.

such proofs are rather inefficient. As we are interested in extracting exact real computation programs, we focus on subsets of Euclidean spaces. For simplicity we present only the one dimensional case here, but generalize that to arbitrary dimension in the implementation. We can prove the following characterization for subsets $A \subseteq \mathbb{R}$ of the real numbers:

$$\mathsf{open}\ A \leftrightarrow \Sigma(F : \mathsf{N} \to \mathsf{R} \times \mathsf{R})).\ (\Pi(n : \mathsf{N}).\ \mathsf{B}(F(n)) \subseteq A) \wedge \Pi(x : \mathsf{R}).\ \exists(n : \mathsf{N}).\ x \in \mathsf{B}(F(n)).$$

Here, $\mathsf{B}(x, r)$ encodes a ball with radius r around x. That is, a subset $A \subseteq \mathbb{R}$ is open if and only if we can find a sequence of balls, all contained in A, that eventually cover all of A.

The proof uses the continuity of the function $\mathsf{R} \to \mathsf{S}$, i.e., the fact that every computable function is continuous. To this end, we need to include a continuity principle in our axiomatic system saying that every function in our type theory is continuous. As the goal of the project is to use axioms to model functionality that is typically available in exact real computation,instead of assuming a continuity principle by saying that any functional $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ is continuous and then reasoning on specific constructions of the types X and Y, decomposing them to the natural numbers, we formalize continuity directly on our axiomatic types in terms of nondeterministic existence of an interval extension. Such an operation is natural in exact real computation software and our continuity principle can be extracted to a simple operation in AERN.

The interval extension can be used to derive a more standard form of the continuity principle:

$$\Pi(f : \mathsf{R} \to \mathsf{S}).\ \Pi(x : \mathsf{R}).\ (f\ x) = \top \to \mathsf{M}\Sigma(n : \mathsf{N}).\ \Pi(y : \mathsf{N}).\ |x - y| < 2^{-n} \to (f\ y) = \top$$

where M is a nondeterminism monad providing logical equivalence to the propositional truncation in the model (see [Xu15] for formulations of similar continuity principles using truncations). Thus, the continuity principle states that for any Sierpinski-valued mapping f from the reals, when $f\ x$ is defined, there nondeterministically exists a natural number n such that f is defined also for any real number y that is 2^{-n}-close to x.

Other classes of sets we consider are compact and overt sets. A subset $A \subseteq X$ is compact if

$$\mathsf{compact}\ A : \Sigma(f : \mathsf{open}\ X \to \mathsf{S}).\ \Pi(U : \mathsf{open}\ X).\ f(U) = \top \leftrightarrow A \subseteq U.$$

A subset $A \subseteq X$ is overt if

$$\mathsf{overt}\ A : \Sigma(f : \mathsf{open}\ X \to \mathsf{S}).\ \Pi(U : \mathsf{open}\ X).\ f(U) = \top \leftrightarrow A \cap U \neq \emptyset.$$

Note that from a computational point of view, the compact subsets are those for which it can be verified that a semidecidable property holds for each point, and the overt subsets are those for which it can be verified that it holds for at least one point.

A class of sets that we are particularly interested in is those that are both compact and overt which we call located subsets. We give another characterization of such sets corresponding to arbitrarily exact drawings. We prove that such sets are closed under affine transformations and under taking limits of fastly converging sequences w.r.t. the Hausdorff metric. As an application we show how to extract programs for generating verified drawings of several types of fractals and generate those drawings for the 2-dimensional case.

References

[Kon21] Michal Konečný. aern2-real: A Haskell library for exact real number computation. `https://hackage.haskell.org/package/aern2-real`, 2021.

[KPT22] Michal Konečný, Sewon Park, and Holger Thies. Extracting efficient exact real number computation from proofs in constructive type theory. *arXiv preprint arXiv:2202.00891*, 2022.

[Pau16] Arno Pauly. On the topological aspects of the theory of represented spaces. *Comput.*, 5:159–180, 2016.

[Xu15] Chuangjie Xu. *A continuous computational interpretation of type theories.* PhD thesis, University of Birmingham, 2015.

Efficient Evaluation for Cubical Type Theories

András Kovács

Eötvös Loránd University
`kovacsandras@inf.elte.hu`

Currently, cubical type theories are the only known systems which support computational univalence. We can use computation in these systems to shortcut some proofs, by appealing to definitional equality of sides of equations. However, efficiency issues in existing implementations often preclude such computational proofs, or it takes a large amount of effort to find definitions which are feasible to compute. In this abstract we investigate the efficiency of the ABCFHL [ABC+21] Cartesian cubical type theory with separate homogeneous composition (hcom) and coercion (coe), although most of our findings transfer to other systems.

Cubical normalization-by-evaluation

In variants of non-cubical Martin-Löf type theory, definitional equalities are specified by reference to a substitution operation on terms. However, well-known efficient implementations do not actually use term substitution. Instead, *normalization-by-evaluation* (NbE) is used, which corresponds to certain *environment machines* from a more operational point of view. In these setups, there is a distinction between syntactic terms and semantic values. Terms are viewed as immutable program code that supports evaluation into the semantic domain but no other operations.

In contrast, in cubical type theories interval substitution is an essential component of computation which seemingly cannot be removed from the semantics. Most existing implementations use NbE for ordinary non-cubical computation, but also include interval substitution as an operation that acts on semantic values. Unfortunately, a naive combination of NbE and interval substitution performs poorly, as it destroys the implicit sharing of work and structure which underlies the efficiency of NbE in the first place. We propose a restructured cubical NbE which handles interval substitution more gracefully. The basic operations are the following.

1. *Evaluation* maps from syntax to semantics like before, but it additionally takes as input an interval environment and a cofibration.

2. *Interval substitution* acts on values, but it has trivial cost by itself; it only shallowly stores an explicit substitution.

3. *Forcing* computes a value to weak head form by sufficiently computing previously stored delayed substitutions.

On canonical values, forcing simply pushes substitutions further down, incurring minimal cost. But on neutral values, since neutrals are not stable under substitution, forcing has to potentially perform arbitrary computation. Here we take a hint from the formal cubical NbE by Sterling and Angiuli [SA21], by annotating neutral values with stability information. This allows us to quickly determine whether a neutral value is stable under a given substitution. When it is stable, forcing does not have to look inside it.

It turns out that there is only a single computation rule in the ABCFHL theory which can trigger interval substitution with significant cost: the coercion rule for the Glue type former. In every other case, only a *weakening* substitution may be created, but all neutral values are stable under weakening, so forcing by weakening always has a trivial cost.

Using canonicity in closed evaluation

In non-cubical type theories, evaluation of closed terms can be more efficient than that of open terms. For instance, when we evaluate an $\mathsf{if}-\mathsf{then}-\mathsf{else}$ expression, we know that exactly one branch will be taken. In open evaluation, the Bool scrutinee may be neutral, in which case both branches may have to be evaluated.

In the cubical setting, *systems of partial values* can be viewed as branching structures which make case distinctions on cofibrations. Importantly, there are computation rules which scrutinize *all* components of a cubical system. These are precisely the homogeneous composition rules (hcom) for strict inductive types. For example:

$$\mathsf{hcom}_{\mathbb{N}}^{r\to r'}\,[\psi \mapsto i.\,\mathsf{suc}\,t]\,(\mathsf{suc}\,b) = \mathsf{suc}\,(\mathsf{hcom}_{\mathbb{N}}^{r\to r'}\,[\psi \mapsto i.\,t]\,b)$$

When we only have interval variables and a cofibration in the context, we do not have to compute every system component to check for suc. In this case, which we may call "closed cubical", we can use the canonicity property of the theory. Here $\mathsf{suc}\,b$ in the hcom base implies that every system component is headed by suc as well. Hence, we can use the following rule instead:

$$\mathsf{hcom}_{\mathbb{N}}^{r\to r'}\,[\psi \mapsto i.\,t]\,(\mathsf{suc}\,b) = \mathsf{suc}\,(\mathsf{hcom}_{\mathbb{N}}^{r\to r'}\,[\psi \mapsto i.\,\mathsf{pred}\,t]\,b)$$

Here, pred is a metatheoretic function which takes the predecessor of a value which is already known to be definitionally suc. The revised rule assumes nothing about the shape of t on the left hand side, so we can compute pred lazily in the output. These lazy projections work analogously for all non-higher inductive types. For higher-inductive types, hcom is a canonical value, so there is no efficiency issue to begin with.

Huber [Hub16, Section 7.2] used a similar definition, but where pred is an internal definition. For general inductive types, we need metatheoretical lazy field projections; for instance, taking the head of a list is not a total internal function, and we need to use the external knowledge that a list is nonempty.

Summary

- Costly interval substitution can only arise from computing with Glue types.
- In closed cubical evaluation, no computation rule forces all components of a system.

We have implemented a system with these properties, and observed large performance improvements over existing systems. We were also able to compute a variant of a Brunerie number definition which is not computable in Cubical Agda. However, many more benchmarks are yet to be adapted, including the original Brunerie number definition.

References

[ABC$^+$21] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. Syntax and models of cartesian cubical type theory. *Math. Struct. Comput. Sci.*, 31(4):424–468, 2021. doi:10.1017/S0960129521000347.

[Hub16] Simon Huber. *Cubical Interpretations of Type Theory.* PhD thesis, University of Gothenburg, 2016.

[SA21] 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. doi:10.1109/LICS52264.2021.9470719.

17

Session 22: Dependently typed programming

Categorical Models of Subtyping

Greta Coraglia[1]* and Jacopo Emmenegger[2]

[1] University of Milan, Via Festa del Perdono, 7, 20122 Milano, IT
greta.coraglia@unimi.it
[2] University of Genoa, Via Dodecaneso, 35, 16146 Genova, IT
emmenegger@dima.unige.it

Abstract

We recover a classical model of dependent type theory and revisit it to describe a version of coercive subtyping, which we provide rules, coherences, and examples of.

1 Introduction

Most categorical models of dependent types have traditionally been heavily set-based: this is the case for categories with families [6], categories with attributes [2, 11], and natural models [1]. In particular, all of them can be traced back to certain discrete fibrations. We extend this intuition to the case of a general, non necessarily discrete, fibration, and use the newly found structure on the fibers to interpret a form of subtyping closely related to coercive subtyping [9].

2 Fibrational Model

A *natural model* is the data of a pair of discrete fibrations $u, \dot{u}$ and a functor Σ such that $u \circ \Sigma = \dot{u}$ and Σ has a right adjoint Δ. In [1, Proposition 1.2] it is shown that this structure is equivalent to the perhaps more widely known notion of category with families (CwF).

$$\begin{tikzcd}
\dot{\mathcal{U}} \arrow[rr, bend right, "\Sigma"'] \arrow[dr, "\dot{u}"'] & & \mathcal{U} \arrow[ll, bend right, "\Delta"'] \arrow[ll, phantom, "\top"] \arrow[dl, "u"] \\
& \mathcal{C} &
\end{tikzcd}$$

Intuitively, the category $\mathcal{C}$ represents contexts and substitutions, the category $\mathcal{U}$ types, the category $\dot{\mathcal{U}}$ terms. Both types and terms are fibered over contexts and, on a given context, $u, \dot{u}$ provide, respectively, a *set* of types or terms. The functor Σ maps each term to its given type, and Δ picks for each type the generic term in that type in the context obtained by extending its context with itself.

We prove that one can suitably extend the structure above to the case where $u, \dot{u}$ are general Grothendieck fibrations – and that the result is equivalent to yet another model appearing in [8], namely comprehension categories.

Definition 2.0.1 (Generalized CwF). *A* generalized CwF *is the data of a pair of fibrations $u, \dot{u}$, a fibration morphism Σ, and a right adjoint Δ to Σ such that unit and counit have cartesian components.*

*This research has been partially funded by the Project PRIN2020 "BRIO - Bias, Risk and Opacity in AI" (2020SSKZ7R) and by by the Department of Philosophy "Piero Martinetti" of the University of Milan under the Project "Departments of Excellence 2023-2027", both awarded by the Ministry of University and Research (MUR).

Generalized CwFs appear with the name of *judgemental dependent type theories* in [4, 5].

Theorem 2.0.2 ([5])**.** *The 2-category of generalized CwFs is equivalent to the 2-category of comprehension categories.*

Now the fiber over a given context is not a *set*, but a *category*, so that we have all the necessary structure to interpret subtyping.

3 Coercive Subtyping

As we have mentioned, our intuition closely follows that of *coercive subtyping*: this is the product of thinking about subtyping as an abbreviation mechanism, meaning that we say that a given type A' is a subtype of A if there is a unique *coercion* from A' to A. Whenever we need a term of type A, then, it suffices to have a term of type A', which we can "plug-in" into A. Coercive subtyping has many computational properties, and throughout the years it has been made to behave very well with other common structures in dependent type theory [3, 10].

Intuitively, we interpret vertical arrows – meaning arrows in $\mathcal{U}, \dot{\mathcal{U}}$ over identities in $\mathcal{C}$ – to coercions: between two given types A, A' there is at most a coercion (up to vertical isomorphism), so that they suitably model abbreviation. In particular, with faithful fibrations, this is precisely unique, and our rules swiftly match the coherence conditions required of coercive subtyping [10, §2.2].

The technical tool we use is that of *comma object*, and we consider what implications it brings to compute it in either the 2-category of functors into $\mathcal{C}$, namely $\mathbf{Cat}_{/\mathcal{C}}$, or in the 2-category of fibrations (in the sense of Grothendieck-Bénabou, [12]) with basis $\mathcal{C}$, denoted $\mathbf{Fib}(\mathcal{C})$. We recover a form of subtyping in the context of these generalized CwFs and describe the corresponding rules.

We are able to interpret judgements of the form

$$\Gamma \vdash a :_f A \quad \text{and} \quad \Gamma \vdash A' \leq_g A,$$

which we can read as, respectively, "as witnessed by f, a is a term of type A in context Γ" and "as witnessed by g, A' is a sub-type of A in context Γ", and are encoded in, respectively,

$$(\Sigma/\mathrm{Id}) \quad \text{and} \quad (\mathrm{Id}/\mathrm{Id}),$$

and prove that our model satisfies rules such as the following *subsumption*.

$$(\text{Sbsm})\ \frac{\Gamma \vdash a : A' \qquad \Gamma \vdash A' \leq_f A}{\Gamma \vdash a : A}$$

We conclude providing a few examples and, in particular, detail what happens in the case that the type fibration u is obtained from the codomain fibration $\mathsf{cod}\colon \mathbf{Set}^2 \to \mathbf{Set}$: this turns out to be a closely related to semantic subtyping as in [7].

References

[1] Steve Awodey. Natural models of homotopy type theory. *Mathematical Structures in Computer Science*, 28(2):241–286, 2018.

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

[3] Gang Chen. Coercive subtyping for the calculus of constructions. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 150–159, New York, NY, USA, 2003. Association for Computing Machinery.

[4] Greta Coraglia and Ivan Di Liberti. Context, judgement, deduction, 2022.

[5] Greta Coraglia and Jacopo Emmenegger. A comparison of type-theoretic comprehensions. forthcoming, 2023.

[6] 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.

[7] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4), sep 2008.

[8] Bart Jacobs. Comprehension categories and the semantics of type dependency. *Theoretical Computer Science*, 107(2):169–207, 1993.

[9] Z Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 02 1999.

[10] Z. Luo, S. Soloviev, and T. Xue. Coercive subtyping: Theory and implementation. *Information and Computation*, 223:18–42, 2013.

[11] Eugenio Moggi. A category-theoretic account of program modules. *Mathematical Structures in Computer Science*, 1(1):103–139, 1991.

[12] Thomas Streicher. Fibred categories à la Jean Bénabou. *arXiv preprint arXiv:1801.02927*, 2022.

LayeredTypes – Combining dependent and independent type systems

Lukas Abelt[1,2] and Alcides Fonseca[1]

[1] LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal
`labelt@lasige.di.fc.ul.pt`, `alcides@ciencias.ulisboa.pt`
[2] Saarland University, Germany

1 Context

Most programmers and computer scientists will be familiar with simple type systems that ensure that the code they write is type safe in the context of the language they are writing it in. However, much more sophisticated type systems exist that can be used to ensure specific properties of a program, such as resource usage (Linear Types [5]), value predicates (Liquid Types [6]), or correct following of a distributed protocol (Session Types [4]).

Some of these type systems can be dependent or independent from each other. Both Liquid Types and Session Types require a base type system with primitive types (such as int, bool) to exist, but they can be orthogonal to each other, as you can have a Liquid Type whose base type is int, and a Session Type that also relies on the base value being int, ignore the liquid predicate.

Given this plethora of type systems, different applications require a different subset of type systems. Not all applications require these advanced type systems (e.g., fully dependent types), trading off static guarantees for (even if short-term) productivity. Untyped languages have been popular in both web applications, scripting and data science. On the other hand, if you are writing a device driver, you will probably want to take advantage of a type system that guarantees that memory usage is constrained to a given bound [3]. Or if you are writing a complex, distributed protocol, you might want to take advantage of Session Types for that part of the program.

In this talk, we try to address the challenge of how multiple type systems can co-exist in the same programming language, allowing different, valid combinations to be used in the type-checking of a program.

To illustrate this in a small example, consider the program in Listing 1 that reads the text contents of a file and prints it to the console line-by-line. In this short snippet, there are multiple concerns to be addressed. Firstly, one wants to ensure that all function calls are performed with variables of the appropriate data types. Secondly, to avoid errors due to an out of bound access, `get` should only be called with an index that does not exceed the list length. Lastly, when working with resources such as a file descriptor it might be useful to track its state and ensure that it is in a proper (closed) state at the end of program execution. Note that the same standard library functions (`createFD`, `readLines`, etc...) may be needed in other programs with different type systems requirements.

These properties can be verified in two major ways.

All features from all type systems can be combined in single, monolithic type system, containing all the complexity that all the different type systems entail. A very similar issue can also be observed when operating on Liquid Types: In certain cases we want to define predicates on orthogonal properties that could be verified independently from one another; However, current implementation of liquid types do not allow such distinction, leading to confusing and unintelligible error messages [2].

Listing 1: Simple example code

```
printLines lines idx len {
  if len != idx then {
    line = get(lines, idx)
    print(line)
    printLines(lines, idx+1, len)
  }
}

filename = "input.txt"
fileDescriptor = createFD(filename)

open(fileDescriptor)

lines = readLines(fileDescriptor)
len = length(lines)
printLines(lines, 0, len)

close(fileDescriptor)
```

Listing 2: Annotations for LayeredTypes

```
-- State Layer definitions
createFD :: state :: {} -> { Closed }
openFile :: state :: {Closed => Open} -> {}
readLines :: state :: {Open => Consumed} -> {}
closeFile :: state :: {Consumed => Closed} -> {}

-- Type layer definitions
get :: types :: List -> int -> string
length :: types :: List -> int
createFD :: types :: string -> FileHandle
open :: types :: FileHandle -> void
readLines :: types :: FileHandle -> List
close :: types :: FileHandle -> void
print :: types :: string -> void
printLines :: types :: List -> int -> int -> void

-- Liquid layer definitions
length :: {List | true} -> {v:int | v>=0}
printLines :: liquid :: { List | true } -> { l:int | l
      >=0 } -> { i:int | i<=l }
get :: liquid :: { List | true } -> { i:int | i<len }

-- State requirement at the end of the program
fileDescriptor :: state :: {Closed}
```

2 Proposed Approach

We propose a second, more principled alternative: LayeredTypes. In LayeredTypes[1], developers can write programs, but can also define additional type systems as layers. Each layer defines the basic types, and how typechecking happens. Additionally, a layer may depend on another layer if it requires information (such as types or typing contexts) from another layer.

Type checking of programs can be partial in the sense that a program may only use some layers (even if the libraries used are defined in more layers), requiring only that each function contains type information for that layer and all dependencies. The missing layers can be added over time, if they are found to be relevant.

We want to note an important distinction to Gradual Typing [8, 7]: Gradual Typing allows to only provide partial typing information in one system, but does not allow to choose different sets of types to verify. It allows to blend between having no types (0) and having types (1). Our proposal supports multiple type systems, with different combinations among them.

We evaluated our approach in a prototype language. Users can provide their own implementations for verification layers and define dependencies between them. In Listing 2 we see a set of annotations that can be added to the base code of Listing 1 to tackle the issues described. We define three separate layers `state`, `types` and `liquid`. Internally, the framework will build a dependency graph thus allowing to verify properties independently from one another where appropriate.

We believe that this layered, incremental approach can help build more powerful and independent type systems while at the same time making it simpler to understand the errors that might arise during verification.

3 Acknowledgments

This work was supported by *Fundação para a Ciência e Tecnologia* (FCT) in the LASIGE Research Unit under the ref. UIDB/00408/2020 and UIDP/00408/2020, by the CMU–Portugal project CAMELOT (LISBOA-01-0247-FEDER-045915), and the RAP project under the reference (EXPL/CCI-COM/1306/2021).

References

[1] Lukas Abelt and Alcides Fonseca. LayeredTypes: A Framework for Layered Type Systems. `https://github.com/LuAbelt/LayeredTypes`, 2023.

[2] Alcides Fonseca, Catarina Gamboa, João David, Guilherme Espada, and Paulo Canelas. Understandable and useful error messages for liquid types. `https://www.youtube.com/watch?v=_SBqwdSFLk8`, 2022.

[3] Martin A. T. Handley, Niki Vazou, and Graham Hutton. Liquidate your assets: Reasoning about resource usage in liquid haskell. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[4] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1), apr 2016.

[5] B.C. Pierce. *Advanced Topics in Types and Programming Languages.* The MIT Press. MIT Press, 2004.

[6] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169. ACM, 2008.

[7] Jeremy Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 2–27, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[8] Satish Thatte. Quasi-static typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, page 367–381, New York, NY, USA, 1989. Association for Computing Machinery.

Towards dependent combinator calculus

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

We have recently given a completely intrinsic presentation of simply typed combinator calculus with extensionality equations and shown the equivalence with simply typed λ-calculus. [2]. The next step is clear: we need to do the same thing for dependent types. We will present below the main ideas of the construction using a shallow embedding in Agda.

In simply typed combinator calculus we have:

```
K : {A B : Set} → A → B → A
K a b = a
S : {A B C : Set} → (A → B → C) → (A → B) → A → C
S f g a = f a (g a)
```

Identity I = S K K is derivable and using the well known abstraction algorithm [4] we can encode all simply typed λ-terms. It is straightforward to come up with dependently typed versions of the combinators:

```
K : {A : Set} {B : A → Set} → (a : A) → B a → A
S : {A : Set} {B : A → Set} {C : (a : A) → B a → Set}
  → ((a : A) (b : B a) → C a b)
  → (g : (a : A) → (B a))
  → (a : A) → C a (g a)
```

Clearly, the non-dependent version arise by instantiating the dependent types with constant families. However, there is a problem when deriving abstraction. E.g. usually we would say

```
λ x → K = K K
```

However, this is no longer correct, because the variable x may occur in the (hidden) types A,B. As a consequence we need to make the type parameters of K and S explicit by reflecting them into terms using a universe:

```
U : Set
El : U → Set
u : U
Π : (A : U) (B : El A → U) → U
```

with the equations El u = U and El (Π A B) ≡ ((a : El A) → El (B a)).

*This work is supported by the Engineering and Physical Sciences Research Council through the grant EP/N028201/1.

For simplicity we use an inconsistent calculus with Type : Type but this could be easily stratified by using universe levels.

We can now redefine the combinators by making the type arguments explicit using the universe:

```
K : {A : U} (B : El A → U) → (a : El A) → El (B a) → El A
S : {A : U} (B : El A → U) (C : (a : El A) → El (B a) → U)
      → ((a : El A) (b : El (B a)) → El (C a b))
      → (g : (a : El A) → El (B a))
      → (a : El A) → El (C a (g a))
```

Now the dependency of the types is a usual term dependency and we can proceed defining bracket abstraction, e.g.

```
λ (x : C) → K = K K-Type (K C) K
```

where K-Type is the term in the universe corresponding to the type of K which can be derived using the combinators (which now include u and Π). However, there is at least one point where we have to appeal to Baron Münchhausen [1]. How do we derive non-dependent K from dependent K? We would like to say:

```
K' : {A B : U} → El A → El B → El A
K' {A} B = K A (K' u B)
```

But here we are using K' in the definition of K'?! It turns out that we can get ourselves out of the swamp by using I as a primitive combinator:

```
I : {A : U} → El A → El A
K' {A} {B} = K (K (K (K (I {u}) u) A) B)
```

Using K' we can derive a non-dependent version of S as well. However, we are still using λ-calculus for polymorphism which eventually needs to be eliinated too.

Summary

The idea is that we start with a type theory with Π-types (without lambda abstraction) and a universe and we are going to develop a dependent combinator calculus in the universe. The main insight is that apart from the dependent version of S and K we now also need u and Π.

We have only started on this work. We need to show that the abstraction algorithm works in general and has all the desired properties. Also we need to give a more semantic argument why the non-dependent version of K can be derived from the dependent one. In the formal version of this construction, we plan to use the initial categories with families [3] with extra structure as our syntax.

We would like to adopt the extensionality axioms to the dependent case. Furthermore the question is whether we can by further application of Münchausian reasoning completely eliminate the outer level and avoid variables altogether as in the non-dependent calculus.

References

[1] Thorsten Altenkirch, Ambrus Kaposi, Artjoms Šinkarovs, and Tamás Végh. The Münchhausen method in type theory. 2022. Submitted to the post-proceedings of TYPES 2022.

[2] Thorsten Altenkirch, Ambrus Kaposi, Artjoms Šinkarovs, and Tamás Végh. Combinatory logic and lambda calculus are equal, algebraically. Submitted to FSCD, 2023.

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

[4] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1986.

18

Session 23: Foundations of type theory and constructive mathematics

Higher Coherence Equations of Semi-Simplicial Types as n-Cubes of Proofs[*]

Hugo Herbelin and Moana Jubert

Université Paris-Cité,
Inria Paris, IRIF, CNRS, France
$\langle$`hugo.herbelin`, `moana.jubert`$\rangle$`@inria.fr`

1 Introduction

The construction of *semi-simplicial types* in type theory as dependent families of types [8] turned out to be remarkably difficult in spite of considerable scrutiny over the past decade (see [7] for an overview). In a proof-relevant setting, the seemingly innocent *semi-simplicial identity* would be witnessed by a family of terms $\alpha_{i,j} : d_i d_j =_{X_n \to X_{n-2}} d_{j-1} d_i$ (for any $0 \leq i < j \leq n$) which may combine together and create new proof terms.

Cube of proofs. Let us illustrate this with the figure opposite. Each face of the cube is associated with a proof term α. There are exactly two ways to deform the lower "meridian" $d_i d_j d_k$ of the cube into the upper one $d_{k-2} d_{j-1} d_i$ relative to X_n, X_{n-3}, either by passing through the backmost faces or through the frontmost ones—both "hemispheres" of the cube. Now, these two ways correspond exactly to the two proofs of the equality $d_i d_j d_k =_{X_n \to X_{n-3}} d_{k-2} d_{j-1} d_i$, given by the compositions $\pi := \alpha_{j,k} * \alpha_{i,k-1} * \alpha_{i,j}$ and $\pi' := \alpha_{i,j} * \alpha_{i,k} * \alpha_{j-1,k-1}$ (up to whiskering). For the equality to be *coherent* at this stage would require to "fill the cube", i.e. exhibit a term $\beta_{i,j,k} : \pi = \pi'$ witnessing the fact that both proofs are indeed equal. Of course the β's would, in turn, fit onto the "3-hemispheres" of a 4-cube and require a term e.g. $\gamma_{i,j,k,\ell}$ to identify their possible compositions—then the process repeats with 5-cubes, 6-cubes, and so on *ad infinitum*.

$$\begin{tikzcd}
 & X_{n-2} \arrow[rr, "d_{k-2}"] & & X_{n-3} \\
X_{n-1} \arrow[ur, "d_{j-1}"] \arrow[rr, dashed, "d_{k-1}"] & & X_{n-2} \arrow[ur, dashed, "d_{j-1}"] & \\
 & X_{n-1} \arrow[uu, dashed, "d_i"] \arrow[rr, dashed, "d_{k-1}"] & & X_{n-2} \arrow[uu, "d_i"] \\
X_n \arrow[uu, "d_i"] \arrow[ur, dashed, "d_j"] \arrow[rr, "d_k"] & & X_{n-1} \arrow[ur, "d_j"] \arrow[uu, dashed, "d_i"] &
\end{tikzcd}$$

Related work. One solution to handle these infinite towers of coherence consists in adding some form of strictness to equality. The first co-author considered, for example, the case of dependent families of *sets* (i.e. 0-truncated types)—see [4], or [5] for a joint work with Ramachandra. Altenkirch, Capriotti and Kraus, on the other hand, extended HoTT with a second equality type which is strict [2]. A whole different approach recently presented at HoTTEST by Kolomatskaia extends type theory with a construction to inhabit dependent streams of types [6].

[*]The authors would like to thank Pierre-Louis Curien as well for his contributions. This abstract has also been submitted to the HoTT/UF 2023 Workshop in Vienna, Austria.

Our approach. We adopt the "n-cubes of proofs" point of view as above. Proof terms fit inside of an infinite collection of n-cubes, and how they may combine together is a consequence of the combinatorial structure of n-cubes. Roughly speaking, an n-cube would be the data of a term $c_n : h^+_{n,n-1} =_{\dots} h^-_{n,n-1}$ identifying both of its $(n-1)$-hemispheres, which are given by composing the $(n-1)$-faces together in some way. The terms $h^+_{n,n-1}, h^-_{n,n-1}$ are proofs of equality themselves, identifying both of the $(n-2)$-hemispheres $h^+_{n,n-2}, h^-_{n,n-2}$. The general picture would be a tower of equalities:

$$c_n : h^+_{n,n-1} =_{\left(h^+_{n,n-2} =_{\left(h^+_{n,n-3} =_{\dots} h^-_{n,n-3}\right)} h^-_{n,n-2}\right)} h^-_{n,n-1}$$

We suspect that examining the exact combinatorial structure[1] of n-cubes and their k-hemispheres will bring us one step closer to understanding all the higher coherence equations necessary to fully construct semi-simplicial types *without the need of univalence*, i.e. internal to MLTT extended with a notion of "infinite streams of data".

2 Hemispheres of the n-Cube

Given some way to encode the faces of an n-cube, a k-hemisphere (relative to the n-cube) is a specific collection of k-faces that "fit nicely together". A first step thus consists in determining what these k-faces might be.

Definition. *We denote by* $(D^n_k, \leq)$ *the poset of* increasing *sequences* $x_1 \dots x_n$ *of length* n *with values* $0 \leq x_i \leq k$. *The order on* D^n_k *is defined* pointwise*, i.e.* $x \leq y \iff \forall\, i,\ x_i \leq y_i$.

The D^n_k's are connected by two notable order-preserving maps: there is a canonical inclusion of D^n_k into D^n_{k+1} which we will write d_*, as well as a map $R : D^n_k \to D^{n+1}_k$ which adds the value k at the end of a sequence. Moreover, we have the following inductive description:

Lemma. $D^n_k = d_* D^n_{k-1} \amalg R D^{n-1}_k$

Our main result so far is the following:

Theorem. *The k-hemispheres of the n-cube are exactly described by D^k_{n-k}. Moreover, all the possible ways to compose the k-faces are given by all the* topological sorts *on D^k_{n-k}.*

We have a working algorithm that generates all the k-faces of both k-hemispheres. It is additionally possible to choose a total order on D^k_{n-k} which is compatible with $\leq$ and gives a canonical choice of topological sort. Several questions that are open:

1. Compute the appropriate whiskering required for composition of k-faces to make sense in type theory.
2. How to deal with the possible choices of composition? Can we show them to be equivalent without the need of new proof terms (e.g. the Eckmann-Hilton argument)?
3. Using the new insights brought by the "n-cubes of proofs" point of view, is it now possible to construct semi-simplicial types in a proof assistant?

We are hopefully close to giving a simple procedure that addresses the first question. This is an ongoing work which may be subject to new developments by the end of June.

[1]Discussions with Métayer revealed that some of this combinatorial structure have already been spelled out over the study of *orientals* (e.g. the work of Aitchison [1]). See [9] for the original article on orientals, and [3] for a recent work of Ara, Lafont and Métayer.

References

[1] Iain R. Aitchison. The geometry of oriented cubes, 2010. Original 1986 report with cosmetic improvements. Available at: https://arxiv.org/abs/1008.1714.

[2] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending Homotopy Type Theory with Strict Equality. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:17. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.CSL.2016.21.

[3] Dimitri Ara, Yves Lafont, and François Métayer. Orientals as free algebras. Preprint, 2022. Available at: http://arxiv.org/abs/2209.08022.

[4] Hugo Herbelin. A dependently-typed construction of semi-simplicial types. *Mathematical Structures in Computer Science*, 25(5):1116–1131, 2015. doi:10.1017/S0960129514000528.

[5] Hugo Herbelin and Ramkumar Ramachandra. A parametricity-based formalization of semi-simplicial and semi-cubical sets. Preprint, 2023. Available at: https://pauillac.inria.fr/~herbelin/articles/nu-types-draft23.pdf.

[6] Astra Kolomatskaia. Semi-Simplicial Types. HoTTEST, December 2022. Available at: https://github.com/FrozenWinters/SSTs.

[7] Nicolai Kraus. On the Role of Semisimplicial Types. Abstract, TYPES 2018. Available at: https://www.cs.nott.ac.uk/~psznk/docs/on_semisimplicial_types.pdf.

[8] Peter LeFanu Lumsdaine. Semi-simplicial types. Online, 2013. Available at: https://ncatlab.org/ufias2012/published/Semi-simplicial+types.

[9] Ross Street. The algebra of oriented simplexes. *Journal of Pure and Applied Algebra*, 49(3):283–335, 1987. doi:10.1016/0022-4049(87)90137-x.

Coinductive control of inductive data types

Paige Randall North[1] and Maximilien Péroux[2]

[1] Utrecht University, Utrecht, Netherlands
`p.r.north@uu.nl`
[2] University of Pennsylvania, Philadelphia, Pennsylvania, United States
`mperoux@sas.upenn.edu`

Abstract

We combine the theory of inductive data types with the theory of universal measurings. By doing so, we find that many categories of algebras of endofunctors are actually enriched in the corresponding category of coalgebras of the same endofunctor. The enrichment captures all possible partial algebra homomorphisms, defined by measuring coalgebras. Thus this enriched category carries more information than the usual category of algebras which captures only total algebra homomorphisms. We specify new algebras besides the initial one using a generalization of the notion of initial algebra.

1 Introduction

In both the tradition of functional programming and categorical logic, one takes the perspective that most data types should be obtained as initial algebras of certain endofunctors (to use categorical language). For instance, the natural numbers are obtained as the initial algebra of the endofunctor $X \mapsto X + 1$, assuming that the category in question (often the category of sets) has a terminal object 1 and a coproduct $+$. Much theory has been developed around this approach, which might be said to culminate in the notion of W-types [2].

In another tradition, that of categorical algebra, algebras (in the traditional sense) over a field k are studied. It has been long understood (going back at least to Wraith and Sweedler, according to [1]) that the category of k-algebras is naturally enriched over the category of k-coalgebras, a fact which has admitted generalization to several other settings (e.g. [1, 3]). Here, we generalize those classic results to the setting of an endofunctor on a category, and in particular those endofunctors that are considered in the theory of W-types.

That is to say, this work is the beginning of a development of an analogue of the theory of W-types – not based on the notion of initial objects in a *category* of algebras, but rather on a generalized notion of initial object in a *coalgebra enriched category* of algebras. The hom-coalgebras of our enriched category carry more information than the hom-sets in the unenriched category that is usually considered in the theory of W-types. We are then able to generalize the notion of *initial algebra*, taking inspiration from the theory of weighted limits, which is more expressive, and thus can be used to specify more objects than the usual notion of initial algebra. Because of our move to the enriched setting, then, we have better control than in the unenriched setting, and we are able to specify more data types than just those which are captured by the theory of W-types.

2 Main results

Our main theorem is the following.

Theorem. *Let* $(\mathsf{C}, \otimes, \mathbb{I}, \underline{\mathsf{C}}(-,-))$ *be a locally presentable symmetric monoidal closed category. Let* $F : \mathsf{C} \to \mathsf{C}$ *be an accessible lax symmetric monoidal endofunctor. Then the category* Alg_F

of F-algebras is enriched, tensored, and powered over the symmetric monoidal category CoAlg_F *of F-coalgebras.*

We show that many endofunctors of interest in the theory of W-types satisfy these hypotheses. For instance, Set is a locally presentable symmetric monoidal closed category. The following functors on a locally presentable symmetric monoidal closed category satisfy the hypotheses: the identity functor, any constant functor at a commutative monoid, the coproduct of two functors that satisfy the hypothesis, and the product two functors that satisfy the hypotheses.

In particular, the functor $X \mapsto X + 1$ on Set satisfies the hypotheses, and we work out very explicitly what the enrichment (and tensoring and cotensoring) tells in this situation. In this concrete case, we see that the enrichment encodes a notion of *partial algebra homomorphism*, whereas the usual category of algebras encodes the notion of *total algebra homomorphism*.

We then observe that there is an implicit parameter in the notion of initial algebra which we may now vary. One might think of an initial object as a certain *colimit*, but in reality, an initial object in a category C is usually (equivalently) defined as an object I with the property that $\mathsf{hom}(I, X) = \{*\}$ for every $X \in \mathsf{C}$. That is, I is the vertex of a cone over the identity functor on C with the special property that each leg of the cone (at an object $X \in \mathsf{C}$) is the only morphism of $\mathsf{hom}(I, X)$. The reader might know that as such, an initial object can always be defined as the *limit* of the identity functor on C. Now that we are in the enriched setting, however, the appropriate notion of limit becomes that of *weighted limit* in which we are able ask not just that $\mathsf{hom}(I, X) = \{*\}$ but that $\mathsf{hom}(I, X) = W$ for any object W. Thus, we make the following definition.

Definition. *Consider a monoidal category* $(\mathsf{C}, \otimes, \mathbb{I}, \underline{\mathsf{C}}(-,-))$ *and endofunctor* $F : \mathsf{C} \to \mathsf{C}$ *satisfying the hypotheses of the above theorem.*

For $W \in \mathsf{CoAlg}_F$*, we define the W-initial algebra to be the limit of the identity functor on* Alg_F *(viewed as the enriched categories described in the above theorem) weighted by the constant functor* $\mathsf{Alg}_F \to \mathsf{CoAlg}_F$ *at W.*

Taking W to be the terminal coalgebra, we recover the notion of initial algebras. But taking alternate W, we can specify many more initial algebras. For instance, considering the endofunctor $X \mapsto X + 1$, we can specify quotients of $\mathbb{N}$ by weighting by subcoalgebras of $\mathbb{N}^\infty$, the terminal coalgebra.

References

[1] Martin Hyland, Ignacio López Franco, and Christina Vasilakopoulou. Hopf measuring comonoids and enrichment. *Proc. Lond. Math. Soc. (3)*, 115(5):1118–1148, 2017.

[2] Per Martin-Löf. Intuitionistic type theory. *Naples: Bibliopolis*, 1984.

[3] Maximilien Péroux. The coalgebraic enrichment of algebras in higher categories. *Journal of Pure and Applied Algebra*, 226(3):106849, 2022.

Read the ~~mood~~mode and stay positive

Lucas Escot[1], Josselin Poiret[2], Joris Ceulemans[3], Andreas Nuyts[3] and Malin Altenmüller[4]

[1] TU Delft, Netherlands
[2] ENS de Lyon, France
[3] imec-DistriNet, KU Leuven, Belgium
[4] University of Strathclyde, Scotland

Languages like Coq and Agda forbid users to define non-strictly-positive data types [AAG05]. Indeed, one could otherwise very easily define non-terminating programs. However, this strict-positivity criterion is nothing more than a syntactic restriction, which prevents sometimes perfectly reasonable and innocuous data types to be defined. We present ongoing work on making positivity checking more modular in Agda, by allowing *polarity annotations* in function types and making it possible to enforce the variance of functions simply by type checking.

The polarity modal system. We introduce a new *modal system* [GKNB21] aptly called the *polarity* [AM04, Abe06] modal system. It consists of a partially-ordered set made out of five *polarities*: `unused`, `-`, `+`, `++` and `mixed`. These elements correspond to the permitted uses of bound variables. The polarities are given the partial order shown on the right, to be read from bottom to top (e.g. `+` $\leq$ `++`) as going from less restrictive to more restrictive: a variable bound with the `mixed` polarity is allowed to appear anywhere; a variable bound with polarity `++` can only appear to the right of arrows; variables bound with polarity `-` (resp. `+`) can appear to the left of an odd (resp. even) number of arrows; and a variable bound with the `unused` polarity can only be used to define constant functions (much like irrelevance). This modal system is given a composition operation $\circ$ whose table we write below:

$$\begin{tikzcd}
& \texttt{unused} \arrow[ddl, dash] \arrow[dr, dash] & \\
& & \texttt{++} \arrow[d, dash] \\
\texttt{-} \arrow[dr, dash] & & \texttt{+} \arrow[dl, dash] \\
& \texttt{mixed} &
\end{tikzcd}$$

$\circ$	mixed	+	++	-	unused
mixed	mixed	mixed	mixed	mixed	unused
+	mixed	+	+	-	unused
++	mixed	+	++	-	unused
-	mixed	-	-	+	unused
unused	unused	unused	unused	unused	unused

$a \circ b$ is to be understood as the *most restrictive polarity* a variable can be bound with, such that it can be used with polarity a in a term that is itself used with polarity b. `unused` is the absorbing element and `++` is the neutral element of this operation. It gives rise to a (left) *division* operation $\backslash$, defined such that $\mu \leq \delta \circ \nu \iff \delta \backslash \mu \leq \nu$ for any δ, μ, ν. The operations $\circ$ and $\backslash$ form a Galois connection.

Typing rules. After attributing a polarity to every variable in context and function domains ($\texttt{@r}\ x : A$), and extending the left-division operation to contexts (${}_{\texttt{r}}\backslash\Gamma$) variablewise, we introduce the typing rules of the polarity modal system. We implicitly use Russel-style universes. Note that the context of the premise of T-EL is left divided by `unused`, which is equivalent to changing all the annotations in the context to `mixed`: informally, variable use in type judgements does not matter.

$$\frac{{}_{\texttt{unused}}\backslash\Gamma \vdash A : \texttt{Set}}{\Gamma \vdash A \text{ type}}\ \textsc{t-El} \qquad \frac{\Gamma \text{ ctx} \qquad \Gamma \vdash A \text{ type}}{\Gamma, \texttt{@r}\ x : A \text{ ctx}}\ \textsc{ctx-Ext} \qquad \frac{\texttt{@r}\ x : A \in \Gamma \qquad \texttt{r} \leq \texttt{++}}{\Gamma \vdash x : A}\ \textsc{t-Var}$$

Modal function types (T-PI) use their domain negatively[1]. Note that left-dividing by `-` corresponds to inverting the annotation's polarity, except for `++` which becomes `unused`. Regardless of the annotation `@r` on the variable x, x is bound in the codomain as `mixed` (the weakest modality). This is in line with T-EL: since usage at the type level does not count, we do not constrain it.

$$\frac{{}_{\texttt{-}}\backslash\Gamma \vdash A : \texttt{Set} \qquad \Gamma, \texttt{@mixed}\ x : A \vdash B : \texttt{Set}}{\Gamma \vdash (\texttt{@r}\ x : A) \to B : \texttt{Set}}\ \textsc{t-Pi} \qquad \frac{\Gamma, \texttt{@r}\ x : A \vdash t : B}{\Gamma \vdash \lambda x.t : (\texttt{@r}\ x : A) \to B}\ \textsc{t-Lam}$$

$$\frac{\Gamma \vdash u : (\texttt{@r}\ x : A) \to B \qquad {}_{\texttt{r}}\backslash\Gamma \vdash v : A}{\Gamma \vdash u\ v : B[v/x]}\ \textsc{t-App}$$

And the monad was free, the (fix)point taken. We implemented the new modal system and the typing rules presented above in Agda[2]. Since other modal systems were readily available — erasure [Dan19, ADV21], irrelevance[3], cohesion [LOPS18] — a lot of the infrastructure was in place. Still, our work highlighted some deficiencies in the current implementation of modalities [NPE+23]. Using our modified version of Agda, the following annotations are valid and taken into account by the type checker.

```
F : @++ Set → Set              G : @- Set → Set              H : @+ Set → Set
F X = Nat → X                  G X = X → Nat                 H X = (X → Nat) → Nat
```

Above, only `F` is *strictly* positive and can be annotated as such without the type checker getting in the way. We extended Agda's positivity checker so that it also uses the polarity of functions during the analysis, allowing the definitions of both the well-known free monad construction `Free` and the least fixed point `Mu` of any strictly positive functor. Note that `F` and `A` could themselves be annotated `++`, we refer to the pull request for more elaborate examples.

```
data Free (F : @++ Set → Set)          data Mu (F : @++ Set → Set) : Set where
    (A : Set) : Set where                In : F (Mu F) → Mu F
  Pure : A              → Free F A
  Free : F (Free F A) → Free F A
```

Next steps. This work is ongoing and much is left to be done. On the semantics side of things, a model for the polarity modalities is still eluding us, especially for the `++` polarity, and it will most likely require looking deeper at directed type theory. The usefulness of our annotations is hindered by the lack of subtyping in Agda, preventing one to use functions of type `@++ Set → Set` wherever `Set → Set` is expected. Even if manual eta-expansion appeases the type checker, a user of our annotation system has to redefine all the usual constructions from scratch. A further complication is the fact that polarity and subtyping can interact in a non-trivial way [Abe06]. Another question left to be answered is whether it is safe to add the primitive `fmap : (F : @+ Set → Set) (f : A → B) → F A → F B` to Agda such that it knows `fmap` is terminating and always reduces as expected. While relaxing the strict-positivity criterion to simply positive data types has been shown to be inconsistent in presence of an impredicative sort [CP88], one can wonder whether it would be safe in Agda [BMS18]. We also want to investigate whether our polarity system could replace Agda's positivity checker entirely, greatly simplifying the implementation and perhaps even improving type checking performance.

[1]As remarked by Nuyts [Nuy15, eqs. (2.43, 2.58 modulo typo)], more care is needed if one wants to take higher categorical structure into account.

[2]https://github.com/agda/agda/pull/6385

[3]The Agda implementation does not seem to follow any specific literature for irrelevance.

Acknowledgements Joris Ceulemans and Andreas Nuyts hold a Phd Fellowship and a Postdoctoral Fellowship (resp.) from the Research Foundation - Flanders (FWO). This research is partially funded by the Research Fund KU Leuven.

We would like to thank the TYPES 2023 reviewers for their comments and suggestions.

References

[AAG05] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theor. Comput. Sci.*, 342(1):3–27, 2005. doi:10.1016/j.tcs.2005.06.002.

[Abe06] Andreas Abel. Polarized subtyping for sized types. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *Computer Science – Theory and Applications*, Lecture Notes in Computer Science, pages 381–392. Springer, 2006. doi:10.1007/11753728_39.

[ADV21] Andreas Abel, Nils Anders Danielsson, and Andrea Vezzosi. Compiling programs with erased univalence. 2021. Draft. URL: https://www.cse.chalmers.se/~nad/publications/abel-danielsson-vezzosi-erased-univalence.pdf.

[AM04] Andreas Abel and Ralph Matthes. Fixed points of type constructors and primitive recursion. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 20-24, 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 2004. doi:10.1007/978-3-540-30124-0_17.

[BMS18] Ulrich Berger, Ralph Matthes, and Anton Setzer. Martin Hofmann's case for non-strictly positive data types. In Peter Dybjer, José Espírito Santo, and Luís Pinto, editors, *24th International Conference on Types for Proofs and Programs, TYPES 2018, June 18-21, 2018, Braga, Portugal*, volume 130 of *LIPIcs*, pages 1:1–1:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.TYPES.2018.1.

[CP88] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988. doi:10.1007/3-540-52335-9_47.

[Dan19] Nils Anders Danielsson. Logical properties of a modality for erasure. 2019. Draft. URL: https://www.cse.chalmers.se/~nad/publications/danielsson-erased.pdf.

[GKNB21] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal dependent type theory. *Log. Methods Comput. Sci.*, 17(3), 2021. doi:10.46298/lmcs-17(3:11)2021.

[LOPS18] 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. doi:10.4230/LIPIcs.FSCD.2018.22.

[NPE$^+$23] Andreas Nuyts, Josselin Poiret, Lucas Escot, Joris Ceulemans, and Malin Altenmüller. Proposal: Multimode Agda, 2023. URL: https://github.com/anuyts/public/blob/1edae3589f9db6f0757699b33f960b1db061e7a4/agda/modal.md.

[Nuy15] Andreas Nuyts. Towards a directed homotopy type theory based on 4 kinds of variance. Master's thesis, KU Leuven, Belgium, 2015.

19

Session 24: Links between type theory and functional programming

Types and Semantics of Extensible Data Types

Cas van der Rest and Casper Bach Poulsen

Technische Universiteit Delft, Delft, The Netherlands

A common litmus test for a language's capability for modularity is whether the programmer is able to both extend existing data with new ways to construct it and add new functionality for this data. All in a way that preserves static type safety; a conundrum which Wadler [14] dubbed the *expression problem*. In the context of pure functional programming further modularity concerns arise from the need to model a program's side effects explicitly using *monads* [8], whose syntax and implementation we would ideally define separately and in a modular fashion.

Traditionally, these modularity questions are tackled in functional languages by embedding the *initial algebra semantics* [4] of inductive data types. This approach was popularized by Swierstra's *Data Types à la Carte* [11] as a solution to the expression problem, and was later applied to modularize the syntax and semantics of both first-order and higher-order effectful computations [7, 15, 10, 12] through various kinds of inductively defined *free monads*. The key idea that unifies these approaches is the use of *signature functors* that act as a syntactic representation of inductive data types or inductively defined free monads, from which we recover the desired structure using a type-level fixpoint. This separation of syntax and recursion permits the composition of data types and effect trees by means of a general co-product of signature functors, an operation that is not available for native data types. However, while embedding signature functors is a tremendously useful technique for enhancing functional languages with a higher degree of (type-safe) modularity, there are still some downsides to the approach.

Problem statement. Since we are working with an embedding of the semantics of data types, we introduce an additional layer of indirection that causes some encoding overhead due to a lack of interoperability with built-in data types. Furthermore, the connection with the underlying categorical concepts that motivate these embeddings remains implicit. By keeping the motivating concepts implicit, our programs lack a rigorously defined formal semantics, but we also introduce further encoding overhead. That is, we usually have to define typeclass instances or work with a *universe construction* [1] to ensure that signatures are indeed functorial.

This work. We advocate an alternative approach that makes the functional programmer's modularity toolkit—e.g., functors, folds, fixpoints, etc.—part of the language's design. We believe that this has the potential to address the issues outlined above. By incorporating these elements into a language's design we have the opportunity to develop more convenient syntax for working with extensible data types (see e.g. the authors' previous work [13]), and by defining a formal semantics we maintain a tight connection between the used modularity abstractions and the concepts that motivate these constructs. The aim of this work is to present a core calculus that acts as a minimal basis for capturing the modularity abstractions discussed here, as well as to develop a formal categorical semantics for this calculus.

Calculus Design and Semantics. We present a (non-dependently typed) λ-calculus with kinds and Hindley-Milner style polymorphism. Types are restricted such that any higher-order type expression is by construction a functor in all its arguments, effectively making the concept of functors first-class in the language's design. By imposing this additional structure, we can provide the programmer with several additional primitives that can be used to capture the aforementioned modularity abstractions, while simultaneously keeping a closer connection to the categorical semantics of these abstractions. Well-formedness of types is defined as usual for the first-order fragment of System F_ω, the only salient difference being that we maintain a separate context, Φ, containing the free variables that a type expression is intended to be functorial in. Type-level λ-abstraction adds a new binding to Φ, and we discard all functorial variables in the domain of a function to enforce that the variables in Φ are only used covariantly:

$$\frac{\Delta \mid \Phi, (X \mapsto k_1) \vdash \tau : k_2}{\Delta \mid \Phi \vdash \lambda X.\tau : k_1 \rightsquigarrow k_2} \qquad \frac{\Delta \mid \emptyset \vdash \tau_1 : \star \qquad \Delta \mid \Phi \vdash \tau_2 : \star}{\Delta \mid \Phi \vdash \tau_1 \Rightarrow \tau_2 : \star}$$

This ensures that all higher-order types have a semantics as objects in an appropriate functor category. The variables in Δ have mixed variance and are bound by universal quantification.

The functor semantics of a type $\tau : k_1 \rightsquigarrow k_2$ guarantees that we can map over values of type τ, provided we have a way to transform the argument type. We expose this ability to the programmer by adding a general mapping primitive to the calculus:

$$\frac{\Delta \mid \emptyset \vdash \tau : k_1 \rightsquigarrow k_2 \qquad \Gamma \vdash M : \tau_1 \xrightarrow{k_1} \tau_2}{\Gamma \vdash \mathbf{map}^{\tau}(M) : \tau\ \tau_1 \xrightarrow{k_2} \tau\ \tau_2} \qquad \begin{array}{c} \sigma \xrightarrow{\star} \tau = \sigma \Rightarrow \tau \\ \sigma \xrightarrow{k_1 \rightsquigarrow k_2} \tau = \forall \alpha.\ \sigma(\alpha) \xrightarrow{k_2} \tau(\alpha) \end{array}$$

We use the syntax $\tau_1 \xrightarrow{k} \tau_2$ to denote a (polymorphic) function that universally closes over all type arguments of τ_1 and τ_2, provided that they have the same kind.

Generally speaking, the intended semantics of a terms is a natural transformation between functors over a bicartesian closed category $\mathcal{C}$. We reify this underlying categorical structure through primitives such as **map**. Other examples of such primitives are operations for destructing fixpoints or co-products:

$$\text{T-FOLD}\ \frac{\Gamma \vdash M : \tau_1(\tau_2) \xrightarrow{k} \tau_2}{\Gamma \vdash \mathbf{fold}^{\tau_1}(M) : \mu(\tau_1) \xrightarrow{k} \tau_2} \qquad \text{T-JOIN}\ \frac{\Gamma \vdash M : \tau_1 \xrightarrow{k} \tau \qquad \Gamma \vdash M : \tau_2 \xrightarrow{k} \tau}{\Gamma \vdash M \blacktriangledown N : \tau_1 \oplus \tau_2 \xrightarrow{k} \tau}$$

To justify these operations we must argue that terms of type $\tau_1 \xrightarrow{k} \tau_2$ represent morphisms in the (functor) category associated with k.

As an example, we compare definitions of the free monad in our calculus (l) and Haskell (r):

$$\mathit{Free} \triangleq \lambda F.\lambda A.\mu X.A \oplus F(X) \qquad \textbf{data}\ \mathit{Free}\ f\ a = \mathit{Pure}\ a \mid \mathit{In}\ (f\ (\mathit{Free}\ f\ a))$$

Free is well-formed with kind $(\star \rightsquigarrow \star) \rightsquigarrow \star \rightsquigarrow \star$. Consequently, it is by construction a functor in both type arguments, guaranteeing that we can always map over either of its arguments:

$$\mathbf{map}^{\mathit{Free}(\gamma)}(f) : \forall\alpha.\forall\beta.\forall\gamma.\mathit{Free}(\gamma)(\alpha) \Rightarrow \mathit{Free}(\gamma)(\beta) \qquad \text{where } f : \alpha \Rightarrow \beta$$
$$\mathbf{map}^{\mathit{Free}}(f) : \forall\alpha.\forall\gamma_1.\forall\gamma_2.\mathit{Free}(\gamma_1)(\alpha) \Rightarrow \mathit{Free}(\gamma_2)(\alpha) \qquad \text{where } f : \forall\alpha.\gamma_1(\alpha) \Rightarrow \gamma_2(\alpha)$$

In Haskell, we would require dedicated instances to witness that *Free* is a (higher-order) functor.

Existing Work. There is some previous work that attacks similar problems [9, 3], but to the best of our knowledge no existing language design can capture the modularity abstractions discussed in this abstract and has a clearly defined categorical semantics. Closest to our work, and a major source of inspiration, is a calculus developed by Johann al. [6, 5] for studying parametricity for nested data types [2]. Still, there are some key differences: in their setting universal quantification is limited to zero-argument types, and the semantics is tied to the category of sets, and relies on an additional interpretation of types as relations.

Conclusion. We have designed a calculus that demonstrates how support for type-safe modularity can be integrated into a programming language's design in a principled way, which we intend as a stepping stone for designing functional languages with better facilities for type-safe modularity. We are finalizing the semantic model that relates this support for modularity to the categorical concepts that motivate it.

References

[1] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.*, 10(4):265–289, 2003.

[2] Richard S. Bird and Lambert G. L. T. Meertens. Nested datatypes. In Johan Jeuring, editor, *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

[3] Yufei Cai, Paolo G. Giarrusso, and Klaus Ostermann. System f-omega with equirecursive types for datatype-generic programming. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 30–43. ACM, 2016.

[4] Joseph A Goguen. An intial algebra approach to the specification, correctness and implementation of abstract data types. *IBM Research Report*, 6487, 1976.

[5] Patricia Johann and Enrico Ghiorzi. Parametricity for nested types and gadts. *Log. Methods Comput. Sci.*, 17(4), 2021.

[6] Patricia Johann, Enrico Ghiorzi, and Daniel Jeffries. Parametricity for primitive nested types. 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 324–343. Springer, 2021.

[7] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 145–158. ACM, 2013.

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

[9] J. Garrett Morris and James McKinna. Abstracting extensible data types: or, rows by any other name. *Proc. ACM Program. Lang.*, 3(POPL):12:1–12:28, 2019.

[10] Casper Bach Poulsen and Cas van der Rest. Hefty algebras: Modular elaboration of higher-order algebraic effects. *Proc. ACM Program. Lang.*, 7(POPL):1801–1831, 2023.

[11] Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008.

[12] Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. Latent effects for reusable language components. In Hakjoo Oh, editor, *Programming Languages and Systems - 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17-18, 2021, Proceedings*, volume 13008 of *Lecture Notes in Computer Science*, pages 182–201. Springer, 2021.

[13] Cas van der Rest and Casper Bach Poulsen. Towards a language for defining reusable programming language components - (project paper). In Wouter Swierstra and Nicolas Wu, editors, *Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers*, volume 13401 of *Lecture Notes in Computer Science*, pages 18–38. Springer, 2022.

[14] Phil Wadler. The expression problem. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, 1998. Accessed: 2020-07-01.

[15] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 1–12. ACM, 2014.

Extensional Equality Preservation in Intensional MLTT

Nuria Brede[1][2], Tim Richter[1], Patrik Jansson[3], and Nicola Botta[2][3]

[1] University of Potsdam, Potsdam, Germany, nuria.brede,tim.richter@uni-potsdam.de
[2] Potsdam Institute for Climate Impact Research, Potsdam, Germany, nicola.botta@pik-potsdam.de
[3] Chalmers University of Technology, Göteborg, Sweden, patrik.jansson@chalmers.se

Dealing with logical independence. There are several important logical principles known to be *independent* of intensional Martin-Löf type theory (ITT), such as *uniqueness of identity proofs (UIP)*, *functional* and *propositional extensionality* and *univalence* (which is incompatible with UIP but entails the two extensionality principles) [1, 2, 3].

When proving a statement in a specific theory, we may distinguish two basic approaches to dealing with such an independent principle: The axiomatic approach – adding the principle as an axiom to the theory; and the local approach – adding assumptions to statements. The axiomatic approach implicitly restricts the models of the theory in which the proof holds, while the local approach restricts the internal applicability of the statement. As an example, consider the choice between either adding UIP to ITT or working with *h-sets* [4] instead of arbitrary types. In the following, we are interested in a principle which has arisen from a local approach to dealing with *functional extensionality*:[1] which as an axiom would read

$$funExt : \{A, B : Type\} \to \{f, g : A \to B\} \to f \doteq g \to f = g$$

(where we write $f \doteq g$ for extensional equality $(a : A) \to f\ a = g\ a$ of the functions f and g, and $(=)$ is the standard intensional equality). Extensional equality is relevant in many contexts. For example when reasoning about generic programs in the style of the Algebra of Programming [6, 7], one is interested in program transformations that preserve the observable behaviour of programs. Our approach treads a middle ground between postulating *funExt* or using a more general *setoid*-based framework [8, 9] like, e.g., [10, 11, 12].[2]

Preservation of extensional equality for functors. For a function $F : Type \to Type$ to be a *functor* (in the usual functional programming sense, considering *Type* as a category with types as objects and functions as morphisms), another function

$$map : \{A, B : Type\} \to (A \to B) \to F\ A \to F\ B$$

is required that preserves identity and function composition. To even state these properties of *map*, one needs a notion of equality of functions. Extensional equality is arguably the most reasonable choice (see e.g. discussions in [14, Sec. 3] and [15]) but straight away leads to consider a further preservation property, namely that *map preserves extensional equality*:

$$mapPresEE : Functor\ F \Rightarrow \{A, B : Type\} \to (f, g : A \to B) \to f \doteq g \to map\ f \doteq map\ g$$

This principle has been used by Matthes [16] in the context of representing nested datatypes in ITT.[3] We have studied the *mapPresEE* principle from a more pragmatic perspective in [14], and advocated for its use(fulness) to reason about generic programs in ITT (see also [17, 18]).[4] For example, we showed that it is crucial to generically relate different representations of monads.

In presence of functional extensionality, *mapPresEE* is uniformly derivable in ITT for any functor. Concrete instances of *mapPresEE* are however also derivable without assuming full-fledged functional extensionality, as shown in [14] (concrete examples can also be found in the standard libraries of Agda and Coq, e.g. for *Maybe* or *List*). Still – as discussed in [14] for the example of the *Reader* functor (with $Reader\ E\ A = E \to A$) and its *map* function

$$mapR : \{E, A, B : Type\} \to (f : A \to B) \to Reader\ E\ A \to Reader\ E\ B$$
$$mapR\ f\ r = f \circ r$$

[1]We are using *Idris* [5] as host language but any other syntax for ITT would do as well.
[2]Yet another approach to dealing with *funExt* "(non-)computationally" (in Agda) is discussed in [13].
[3]We are very grateful to the reviewer who pointed us to Matthes' usage of *mapPresEE* that we had been unaware of.
[4]The sources of [14] can be found at [19], where we will also upload updates on the current development.

– there are also functors for which proving *mapPresEE* without *funExt* seems (and is indeed, see below) impossible. And yet, for *Reader*, a variant of *mapPresEE* for a two-argument version of extensional equality

$$
\begin{aligned}
&(\ddot{=}) : \{A, B, C : \mathit{Type}\} \to (f, g : A \to B \to C) \to \mathit{Type} \\
&(\ddot{=})\ f\ g = (a : A) \to f\ a \doteq g\ a \\
&\mathit{mapRPresEE2} : \{E, A, B : \mathit{Type}\} \to (f, g : A \to B) \to f \doteq g \to \mathit{mapR}\ f \ddot{=} \mathit{mapR}\ g \\
&\mathit{mapRPresEE2}\ f\ g\ \mathit{fEEg}\ r\ a = \mathit{fEEg}\ (r\ a)
\end{aligned}
$$

is easily derivable (which, however, rather is a *transformation* between two notions of equality than a *preservation*). This suggests that exploring the interaction of *map* with extensional equality allows to make fine-grained distinctions between classes of endofunctors on *Type*, and leads us to ask:

- Can we characterise the class of functors for which *mapPresEE* holds?
- Can we prove that this class is a proper subclass of the class of all functors, i.e., that *mapPresEE* is independent from ITT? Can we make its relation to *funExt* more precise?
- How do analogues of the *mapPresEE* principle for lifted versions of extensional equality interact with functional extensionality?

We are currently working on these questions and can give some preliminary answers.

Characterising *mapPresEE*-functors. The second author has proved that for any $T : \mathit{Type} \to \mathit{Type}$ which is a *Traversable* functor [20, 21], i.e. can be equipped with a function

$$
\begin{aligned}
\mathit{traverse} : (G : \mathit{Type} \to \mathit{Type}) \to \mathit{Applicative}\ G \Rightarrow \\
\{A, B : \mathit{Type}\} \to (A \to G\ B) \to T\ A \to G\ (T\ B)
\end{aligned}
$$

satisfying some properties, *mapPresEE* is derivable in ITT with η-equality. The basic observation is that the type of pairs of extensionally equal functions from A to B, $\Sigma\ (A \to B, A \to B)\ (\lambda p \Rightarrow \pi_1\, p \doteq \pi_2\, p)$, is isomorphic to the type of functions from A to $B^{=}$, the *free path space* (cf. [22, Remark 1.12.1]) over B. The free path space construction $(\text{-})^{=}$ (trivially) is an *Applicative* [20] functor and *traverse* $(\text{-})^{=}$ proves *mapPresEE*.

Independence of *mapPresEE* from ITT and relation to *funExt*. Revisiting the *Reader* example above, it has been observed by the first author that a first level *mapPresEE* for *mapR* which is polymorphic in *Reader*'s first argument E is logically equivalent to *funExt*. If *Reader* is specialised to a particular type E, the principle is logically equivalent to a local version of functional extensionality for functions with domain E. (Both results require η-equality.) Similar results can be obtained for variants of *map* for indexed functions or contravariant functors.

Principles for lifted equalities. We have seen that for *Reader* and *mapR*, a principle similar to *mapPresEE* is provable using a notion of extensional equality for functions with two arguments. The lifting of a binary relation can be done systematically by iterating the function

$$
\begin{aligned}
&\mathit{extify} : \{A, B : \mathit{Type}\} \to (B \to B \to \mathit{Type}) \to ((A \to B) \to (A \to B) \to \mathit{Type}) \\
&\mathit{extify}\ \{A\}\ \mathit{relB}\ f\ g = (a : A) \to \mathit{relB}\ (f\ a)\ (g\ a)
\end{aligned}
$$

such that $(\doteq)$ corresponds to *extify* $(=)$, $(\ddot{=})$ corresponds to *extify* $(\doteq)$, and so on. The *Reader* example we have seen above suggests that there is an interaction between the number of function arguments and the "level" of extensional equality for which a *mapPresEE*-like principle is provable. Indeed, analogues of *mapPresEE2* above are provable for n-argument versions of *Reader* with arbitrary $n : \mathbb{N}$ (i.e. n-argument function types). In the presence of product/Σ-types we can reduce an n-argument function to a 1-argument function via uncurrying. But since the interaction of extensional equality with the function space constructor has a distinctively syntactical flavour, it still seems worthwhile to consider generalisations for $n : \mathbb{N}$ (with context types $E_1, ..., E_n$ left implicit)

$$
\mathit{mapRTransfEE}_n : \{A, B : \mathit{Type}\} \to (f, g : A \to B) \to f \doteq g \to \mathit{mapR}_n\ f \overset{n}{=} \mathit{mapR}_n\ g
$$

such that *mapPresEE* for *mapR* arises as $\mathit{mapRTransfEE}_1$ (with $(\overset{n}{=})$ as n-th iteration of *extify*). Taking a closer look at this family, we see that $\mathit{mapRTransfEE}_0$ is equivalent to *funExt*. Instances for $n > 0$ only prove "localised" versions of function extensionality.

References

[1] Thomas Streicher. *Semantics of type theory - correctness, completeness and independence results.* Progress in Theoretical Computer Science. Birkhäuser, 1991.

[2] Martin Hofmann and Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. In *Proc. Symposium on Logic in Computer Science (LICS '94)*, pages 208–212, 1994.

[3] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *ACM SIGPLAN Conf. on Certified Programs and Proofs*, CPP 2017, pages 182–194. ACM, 2017.

[4] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath — a computer-checked library of univalent mathematics. Available at `https://github.com/UniMath/UniMath`, 2021.

[5] Edwin Brady. *Type-Driven Development in Idris.* Manning Publications Co., 2017.

[6] Richard S. Bird and Oege de Moor. *Algebra of programming.* Prentice Hall International Series in Computer Science. Prentice Hall, 1997.

[7] Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. Algebra of programming in Agda: dependent types for relational program derivation. *Journal of Functional Programming*, 19(5):545–579, 2009.

[8] Errett Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, 1967.

[9] Martin Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, University of Edinburgh, 1995.

[10] Bas Spitters and Vincent Semeria (Maintainers). Coq Repository at Nijmegen (Version 1.2.0). `https://github.com/coq-community/corn`, 2017.

[11] Fredédéric Blanqui and Adam Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859, 2011.

[12] Jason Z. S. Hu and Jacques Carette. Formalizing category theory in Agda. In *ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2021, pages 327–342, New York, NY, USA, 2021.

[13] Chuangjie Xu. Using function extensionality in agda, (non-)computationally. `https://cj-xu.github.io/talks/xu-funext.pdf`, 2017.

[14] Nicola Botta, Nuria Brede, Patrik Jansson, and Tim Richter. Extensional equality preservation and verified generic programming. *Journal of Functional Programming*, 31:e24, 2021.

[15] Nicholas Drozd et al. Proposal: Verify algebra interfaces by default. Discussion on idris-lang `https://groups.google.com/forum/#!topic/idris-lang/VZVpi-QUyUc`, 2020.

[16] Ralph Matthes. An induction principle for nested datatypes in intensional type theory. *Journal of Functional Programming*, 19(3-4):439–468, 2009.

[17] Nuria Brede and Nicola Botta. On the correctness of monadic backward induction. *Journal of Functional Programming*, 31:e26, 2021.

[18] Patrik Jansson. FP talk on Extensional equality preservation. `https://www.youtube.com/watch?v=tvpjuQyPYXI`, 2020.

[19] Nicola Botta et al. Sources and accompanying code of recent papers. `https://gitlab.pik-potsdam.de/botta/papers`, 2020–2023.

[20] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, January 2008.

[21] Mauro Jaskelioff and Ondrej Rypacek. An investigation of the laws of traversals. In James Chapman and Paul Blain Levy, editors, *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012*, volume 76 of *EPTCS*, pages 40–49, 2012.

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

Composable partial functions in Coq, totally for free

Théo Winterhalter

Inria, France

I propose an early-stage Coq [Coq23] library[1] for general recursion. The goal is to be able to able to write functions that may be only partially defined and still prove things about them. This work follows in the footsteps of 'Turing-completeness totally free'[2] [McB15] and the Braga method [LM21]. Indeed, we use a free monad to represent general recursion that is close to the former and we can instantiate it to total functions that can be extracted using ideas similar to the latter. Crucially, this library supports composition of such functions at virtually no cost.

Let us look at the simple example of integer division `div` (`n`, `m`) which returns $\lceil \frac{n}{m} \rceil$ by counting the number of times it can subtract `m` to `n` before reaching 0. There are two obstacles to writing this function directly in Coq: (1) this function is not defined when `m` = 0 ; (2) Coq will fail[3] to see that `n` − `m` is structurally smaller than `n` when checking that the function is terminating. So, instead of defining some function of type ∀ (`n m` : ℕ), ℕ we can use the library's partial function type ∇ (`p` : ℕ ∗ ℕ), ℕ. Below, we define the function in a rather straightforward way, using the Equations plugin [SM19a; SM19b] to get a nice syntax.

```
Equations div : ∇ (p : ℕ ∗ ℕ), ℕ :=
  div (0, m) := ret 0 ;
  div (n, m) := q ← rec (n − m, m) ;; ret (S q).
```

Aside from the explicit monadic operations `ret` and `rec`, this should look as expected. `ret` is just the monadic return that lets you. . . return a value. `rec` is the operation that lets you perform a recursive call, the second branch is thus to be understood as `S` (`div` (`n` − `m`, `m`)).

Compositionality. One of problems found in literature is the lack of compositionality. With this library it is seamless as you can see in the following (silly) function for testing divisibility.

```
Equations test_div : ∇ (p : ℕ ∗ ℕ), 𝔹 :=
  test_div (n, m) := q ← call div (n, m) ;; ret (q ∗ m =? n).
```

Herein, we use the `call` operation that is similar to `rec` but additionally expects a partial function. In fact, `call` is quite flexible in term of partial functions: `f` can be called as long as the class `PFun f` can be inferred. This class records various informations such as the domain, codomain, *etc.* of `f`, *i.e.* all the information required to construct actual (total) functions. What makes this example well typed is that for each `f` : ∇ (`x` : `A`), `B` we have an instance of `PFun f`. This `call` operator takes away the need to do complex encodings and makes it customisable (nothing prevents us *e.g.* to declare an instance of `PFun` for a total function).

Partial functions at work. Of course, merely describing partial functions isn't satisfactory in itself. We want to be able to *run* them, be it within Coq itself, or after extraction. All of this we get *for free*, once and for all. From a user perspective we simply have to use combinators. For instance, we can write `div` @ (10, 5) to apply `div` to 10 and 5 and obtain a value of type ℕ. Note the absence of `option` or any possibility of failure, we got ourselves a natural number. Of course the function did not magically become total or started returning garbage. If we were to

[1]Available at https://github.com/TheoWinterhalter/coq-partialfun/tree/types2023.
[2]Any resemblance to this title is entirely coincidental.
[3]And rightly so when `m` = 0.

write `div @ (10, 0)` instead, we would simply get an error from Coq, it wouldn't be well typed. So, how does this work? Under the hood we actually use two interpretations of `div`:

`fueled div : ∀ (n : ℕ) (p : ℕ * ℕ), Fueled ℕ` `def div : ∀ (p : ℕ * ℕ), domain div p → ℕ`

`fueled` produces a version that performs structural recursion on its first argument `n`, called fuel. It represents the depth of recursive calls that can be performed when evaluating the partial function. If the fuel is exhausted before getting to a value, the whole function returns `NotEnoughFuel`, otherwise it returns `Success v` for some value `v`. `def` on the other hand always returns a value in ℕ, but its application is guarded by a proof that its argument is in the domain of the function. We define it as `domain f x := ∃ v, graph f x v` where the graph is defined as an inductive predicate (thus posing no termination problem).

Now, the trick to be able to write `div @ (10, 5) : ℕ` is to call `def div` with a proof that (10, 5) is its domain *computed* from `fueled div (10, 5)`, or rather from the fact that it is a `Success`. It however has its limitations because we pick a value of fuel once and for all, and besides this will only work for concrete values for the argument and not for variables.

Functional induction. Executing partial functions is good, but we also want to be able to reason about such programs. To this end we provide a functional induction predicate as follows:

`funind : (∇ (x : A), B) → (A → Prop) → (∀ x, B x → Prop) → Prop`

so that `funind f P Q` states that for all input `x` verifying `P x`, corresponding outputs `v` will verify `Q x v`. This fact is proven about the graph of f, but also about `fuled f` and `def f`. For instance we can prove about `div` that it preserves the invariant that inputs (`n`, `m`) with `n` < `m` return 0 if `n` = 0 and otherwise `1`.

`funind div (λ '(n, m), n < m) (λ '(n, m) q, match n with 0 ⇒ q = 0 | _ ⇒ q = 1 end).`

Proving this lemma requires no internal knowledge of the library which contributes to the aimed 'for-free' experience. Similarly, the library provides a way to easily prove `domain` without exposing its internals: instead of reasoning on the graph, we can simply reason about recursive calls. For `div` it becomes easy to prove ∀ `n m`, (`n` = 0 ∨ `m` ≠ 0) ↔ `domain div (n, m)`.

Related work. This line of work is of course not new. I already mentioned the work of McBride [McB15] who proposes a dependent general recursion monad that is very close to this one, except it does not readily support composition to the best of my knowledge. Prior work from Bove and Capretta [BC05] also suffered from this issue, but already laid the grounds for representing partial functions as total functions in Agda. Larchey-Wendling and Monin [LM21] went along the same direction, further proposing ways to make such programs suitable for extraction. In a sense, my proposal is a *composition* of these works by essentially applying the Braga method to a program written in the general recursion monad, as well as an extension to support composition of functions in a neat way.

Future work. My initial motivation for this library is to be able to be able to prove properties about the type checker of Coq written in the MetaCoq project [Soz+20b; Soz+20a] without having to rely on a strong normalisation axiom. Especially when this axiom might be violated by further extensions of (Meta)Coq such as user-defined rewrite rules.

Another direction I find very interesting and which was already proposed by Larchey-Wendling and Monin [LM21] would be to integrate this machinery directly within Equations. This would make it possible to hide the monad and make for an actual seamless experience.

References

[BC05] Ana Bove and Venanzio Capretta. "Modelling general recursion in type theory". In: *Mathematical Structures in Computer Science* 15.4 (2005), pp. 671–708.

[Coq23] Coq development team. *The Coq proof assistant reference manual.* Version 8.17. LogiCal Project. 2023. URL: http://coq.inria.fr.

[LM21] Dominique Larchey-Wendling and Jean-François Monin. "The Braga Method: Extracting Certified Algorithms from Complex Recursive Schemes in Coq". In: *Proof and Computation II.* WORLD SCIENTIFIC, Aug. 2021, pp. 305–386. DOI: 10.1142/9789811236488_0008. URL: https://hal.inria.fr/hal-03338785.

[McB15] Conor McBride. "Turing-completeness totally free". In: *Mathematics of Program Construction: 12th International Conference, MPC 2015, Königswinter, Germany, June 29–July 1, 2015. Proceedings 12.* Springer. 2015, pp. 257–275.

[SM19a] Matthieu Sozeau and Cyprien Mangin. *Equations Reloaded Accompanying Material.* Available on the ACM DL. Aug. 2019. DOI: 10.1145/3342526.

[SM19b] Matthieu Sozeau and Cyprien Mangin. *Equations v1.2.* May 2019. DOI: 10.5281/zenodo.3012649. URL: https://doi.org/10.5281/zenodo.3012649.

[Soz+20a] Matthieu Sozeau et al. "Coq Coq correct! Verification of type checking and erasure for Coq, in Coq". In: *PACMPL* 4.POPL (2020). DOI: 10.1145/3371076.

[Soz+20b] Matthieu Sozeau et al. "The MetaCoq Project". To appear in Journal of Automated Reasoning. 2020. URL: https://hal.inria.fr/hal-02167423.

Towards a Translation from Liquid Haskell to Coq

Lykourgos Mastorou[1], Niki Vazou[1], and Michael Greenberg[2]

[1] IMDEA Software Institute, Spain [2] Stevens Institute of Technology, USA

Liquid Haskell [7] is a refinement type checker for Haskell programs that can also be used as a theorem prover to mechanically check user-provided proofs. For example, it has been used to mechanize proofs about equational reasoning [6], relational properties [5], and program security [3]. These case studies demonstrate that mechanically checking theorems using Liquid Haskell is possible, but they illustrate two main disadvantages. First, the foundations of Liquid Haskell as a theorem prover have neither been well studied nor formalized. Second—and more pressingly—although Liquid Haskell can check proofs, it does not assist in their development. That is, when a fully automatic the proof fails, the proof engineer can't directly inspect the proof at the point of failure—making it difficult to further develop the proof. We can address both disadvantages at once by translating Liquid Haskell proofs to Coq. To understand this translation, we started by encoding the Ackermann function and related arithmetic theorems in Liquid Haskell and in (to-be-automatically-derived) Coq. In this abstract, we present how we aim to translate the **four** main ingredients of Liquid Haskell functions and proofs.

1. Refinement Types $\Longrightarrow$ Subset Types Refinement types are types refined with logical predicates. For example, `Nat` $\doteq \{v : \texttt{Int} \mid 0 \leq v\}$ is the type of integers refined to be natural numbers. Liquid Haskell's refinements are dependent types: $n : \texttt{Int} \rightarrow \{v : \texttt{Int} \mid n \leq v\}$ is the type of a function that increases its integer argument. We could encode refinement types in Coq two ways: inductive predicates or subset types. While Coq works more easily with inductive predicates, they are not a good choice for us: they do not permit "breaking the invariants". Suppose $n : \texttt{Nat}$ and $m : \{v : \texttt{Int} \mid 1 \leq v\}$. Refinement types can easily show that $(n-1)+m : \texttt{Nat}$, even though $n-1$ can be negative. In order to separate values and operations from their properties, we encode refinement types as subset types.

In Coq, an element of the subset type $\{v : \texttt{b} \mid p\ v\}$ is a pair (e, q) of an expression e and a proof that $p\ e$ holds. In Liquid Haskell, an element of a subset type is just the expression e. Our translation's primary challenge is to fill in the proof terms.

2. Reflection & Termination Metrics $\Longrightarrow$ Equations & Induction Principles Liquid Haskell uses a termination checker to ensure user defined functions terminate—Liquid Haskell can only reason safely about terminating functions. When termination is not structurally obvious, termination metrics let us check semantic termination. For example, the metric `/ [m,n]` below expresses that `ack m n` should use lexicographic ordering on its arguments; Liquid Haskell will check that the metric is well founded.

```
{-@ ack :: m:Nat -> n:Nat -> Nat / [m,n] @-}
ack 0 n = n + 1
ack m n = if n == 0 then ack (m-1) 1 else ack (m-1) (ack m (n-1))
```

Terminating functions can be *reflected* [8] in the refinement logic. The annotation `{-@ reflect ack @-}` reflects the Ackermann function, which in practice means that `ack` can appear in the refinements and that the logic "knows" the function's definition—so we can more easily write and prove theorems about `ack`. For example, the theorems below state that `ack` is monotonic; their (omitted) proofs are Haskell functions that inhabit the types. (The type $\{p\}$ is really $\{v : () \mid p\}$, a refinement of unit used as a notion of proposition.)

```
{-@ monotonic_one :: m:Nat -> n:Nat -> {ack m n < ack m (n+1)} @-}
{-@ monotonic :: m:Nat -> n:Nat -> p:{Nat | p < n } -> {ack m p < ack m n} / [n] @-}
```

We use Coq's `Equations` [4] to define functions that are not obviously terminating. Coq's `Fixpoint` only permits structural induction, while `Program Fixpoint` provides opaque functions. We use `Equations` to encode functions like Ackermann's in Coq, yielding both the function's definition and its `Equations`-generated induction principle.

3. Implicit Semantic Subtyping $\Longrightarrow$ Custom Tactics Liquid Haskell implicitly uses subtyping to weaken the types of expressions to their appropriate subtypes. Such subtyping occurs in two program locations: join points and function applications. For example, to type `if p then 2 else 4` as `Nat`, the singleton branch types $\{v : \texttt{Int} \mid v = 2\}$ and $\{v : \texttt{Int} \mid v = 4\}$ will both be weakened to `Nat` via implicit subtyping. Similarly, typing of `f 4`, where $f : \texttt{Nat} \to \texttt{Int}$, succeeds because of implicit subtyping in the argument.

We created `ref_tacts`, a new suite of tactics, to simulate Liquid Haskell's implication checking. At join points, we use the `my_trivial` tactic emulates what is trivial for Liquid Haskell's logic, including destruction and `lia`'s arithmetic. At function applications, the `reft_pose` tactic does a bit more: (1) it grabs the function's preconditions, (2) it proves that the arguments satisfy them, and, critically (3) it makes the proof term inside the argument opaque so that it does not clutter the proof environment. Note that functions arguments have correct subset types—but those types may not be the function's domain type. Part of `reft_pose`'s job is to 'upcast' arguments to satisfy the function domain's preconditions.

4. SMT & Proof by Logical Evaluation (PLE) $\Longrightarrow$ Sniper How do we define `ref_tacts`? Liquid Haskell resolves semantic subtyping's implications by an SMT solver. SMT solvers know all kinds of things—notably, linear arithmetic. Consider this proof of `monotonic`:

```
monotonic m n p | n == p + 1 = monotonic_one m p
                | otherwise  = ack m p     ? monotonic m (n-1) p
                           =<< ack m (n-1) ? monotonic_one m (n-1)
                           =<< ack m n     *** QED
```

The goal is to prove that `ack m p < ack m n`. In the base case, where `n == p + 1`, the proof concludes by `ack m p < ack m (p+1)`. In the inductive case, we use Liquid Haskell's combinators to build the proof. First, we call the inductive hypothesis `monotonic m (n-1) p` to find `ack m p < ack m (n-1)`. Next, `monotonic_one` lets us find `ack m (n-1) < ack m n`. The proof concludes by linear arithmetic and transitivity of `(<)`, which SMT knows.

Using `lia` to translate the proof above to Coq is easy. Unfortunately, we must explicitly use transitivity of `<`... even though transitivity is 'free' in SMT!

Proof translation becomes still more challenging *Proof by Logical Evaluation* [8] (PLE). PLE evaluates expressions in the SMT solver itself, substantially shrinking Liquid Haskell proofs—one need only invoke lemmas. For example, with PLE, the inductive case of `monotonic` could be `monotonic m (n-1) p ?  monotonic_one m (n-1)`. Translating this simpler proof calls for proof search, since intermediate term for transitivity (`ack m (n-1)`) has vanished.

Happily, the recently developed `Sniper` [2] tactic gracefully combines both SMT knowledge and proof search. Sniper provides general proof automation and combines `SMTCoq` [1] with general Coq tactics. We conjecture that `sniper` would be ideal for our Liquid Haskell to Coq translation. In order to apply Sniper in our setting, we must extend it to support equations and subset types that—critical parts of our translation.

Conclusion We aim to translate Liquid Haskell to Coq. Early experiments have produced workable translations of types, functions, subtyping, and Liquid Haskell's refinement logic. `Sniper` [2] offers a promising way forward, once it can reason properly about subset types.

Acknowledgments

We thank Aliaksandr Hryzlou and Chantal Keller for their feedback and discussions. This work is partially founded by the Horizon Europe ERC Starting Grant CRETE (GA: 101039196), the Atraccin de Talento grant (Ref. 2019-T2/TIC-13455), and the Juan de la Cierva grant (IJC2019-041599-I).

References

[1] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of sat/smt solvers to coq through proof witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, pages 135–150, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Valentin Blot, Louise Prisque, Chantal Keller, and Pierre Vial. General automation in coq through modular transformations. 07 2021.

[3] James Parker, Niki Vazou, and Michael Hicks. Lweb: Information flow security for multi-tier web applications. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[4] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in coq. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019.

[5] Elizaveta Vasilenko, Niki Vazou, and Gilles Barthe. Safe couplings: Coupled refinement types. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022.

[6] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Theorem proving for all: Equational reasoning in liquid haskell (functional pearl). *SIGPLAN Not.*, 53(7):132144, sep 2018.

[7] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 269282, New York, NY, USA, 2014. Association for Computing Machinery.

[8] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: Complete verification with smt. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

Reviewers

Stefano Berardi
Marc Bezem
Ulrik Buchholtz
Davide Castelnovo
Evan Cavallo
Cyril Cohen
Pietro Di Gianantonio
Jacopo Emmenegger
José Espírito Santo
Giulio Fellin
Zeinab Galal
Álvaro García-Pérez
Herman Geuvers
Silvia Ghilezan
Daniel Gratzer
Robert Harper
Eduardo Hermo Reyes
Furio Honsell
Ambrus Kaposi
Delia Kesner
Dominik Kirst
Ekaterina Komendantskaya
Neelakantan Krishnaswami
Marina Lenisa
Alexandre Madeira
Assia Mahboubi
Ralph Matthes
Sara Negri
Jovana Obradović
Luca Padovani
Iosif Petrakis
Frank Pfenning
Luís Pinto
Pierre-Marie Pédrot
Ivan Scagnetto
Anton Setzer
Filip Sieczkowski
Matthieu Sozeau
Matteo Tesi
Dominik Wehr
Tom de Jong
Benno van den Berg