EPTCS 396

Proceedings of the

18th International Workshop on

Logical Frameworks and Meta-Languages: Theory and Practice

Rome, Italy, 2nd July 2023

Edited by: Alberto Ciaffaglione and Carlos Olarte

Published: 17th November 2023
DOI: 10.4204/EPTCS.396
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Table of Contents i

Preface ii

Invited Paper: On the Practicality and Soundness of Refinement Types 1
Niki Vazou

Contextual Refinement Types 4
Antoine Gaulin and Brigitte Pientka

Semi-Automation of Meta-Theoretic Proofs in Beluga 20
Johanna Schwartzentruber and Brigitte Pientka

Parallel Verification of Natural Deduction Proof Graphs 36
James T. Oswald and Brandon Rozek

An Interpretation of E-HA^w inside HA^w 52
Félix Castro

Preface

This volume contains a selection of papers presented at LFMTP 2023, the 18th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice. The workshop, affiliated with the 8th International Conference on Formal Structures for Computation and Deduction (FSCD), was held in Rome, Italy, on July 2, 2023. We are very grateful to the organization of FSCD for providing the infrastructure and coordination with other events.

Logical frameworks and meta-languages form a common substrate for representing, implementing and reasoning about a wide variety of deductive systems of interest in logic and computer science. Their design, implementation and their use in reasoning tasks, ranging from the correctness of software to the properties of formal systems, have been the focus of considerable research over the last two decades. This workshop brings together designers, implementors and practitioners to discuss various aspects impinging on the structure and utility of logical frameworks, including the treatment of variable binding, inductive and co-inductive reasoning techniques and the expressiveness and lucidity of the reasoning process.

For this edition, we received 5 submissions, which were reviewed by at least three members of the program committee. After thorough evaluation, the program committee decided to accept the 5 papers for presentation and selected 4 for inclusion in the present EPTCS volume.

We want to express our sincere thankfulness to all the authors who submitted papers to the workshop. We would also like to acknowledge the exceptional efforts of the program committee, that ensured the high quality of the event. The program committee was formed by Roberto Blanco (MPI-SP), Frédéric Blanqui (Inria), Ana Bove (Chalmers University of Technology), Amy Felty (University of Ottawa), Assia Mahboubi (Inria), Narciso Martí-Oliet (Universidad Complutense de Madrid), Gopalan Nadathur (University of Minnesota), Clément Pit-Claudel (Amazon AWS), Andrei Popescu (University of Sheffield), and Claudio Sacerdoti Coen (University of Bologna). We extend our appreciation to the external reviewers, Théo Winterhalter and Samuele Maschio, for their detailed comments engaging in fruitful discussions.

Lastly, we would like to convey to Niki Vazou our deep gratitude for her invaluable contribution as the invited speaker at LFMTP 2023.

October 04, 2023

Alberto Ciaffaglione and Carlos Olarte
PC chairs of LFMTP 2023

A. Ciaffaglione and C. Olarte (Eds.): Logical Frameworks
and Meta-Languages: Theory and Practice (LFMTP 2023)
EPTCS 396, 2023, pp. ii–ii, doi:10.4204/EPTCS.396.0

On the Practicality and Soundness of Refinement Types

Niki Vazou
IMDEA Software Institute
Madrid, Spain
niki.vazou@imdea.org

Refinement types [5, 3] are a software verification technique that extends types of an existing programming language with logical predicates, to verify critical program properties not expressible by the existing type system. As an example, consider the type of the list indexing function `(!!)` that takes as input a list `xs` and an index `i` and returns the `i`th element of `xs`.

```
Existing Type : (!!) :: [a] → Int → a
 Refined Type : (!!) :: xs:[a] → i:{Int | 0 ≤ i < len xs} → a
```

Here, the two arguments are named and the logical predicate `0 ≤ i < len xs` is used to refine the type of the index expressing safe indexing. With this refinement type, each time list indexing is used, i.e., `xs !! i`, the refinement type checker will check that the index `i` is within bounds.

1 Refinement Types are Designed to be Practical

Refinement type are *naturally integrated* in existing programming languages. For example, in Liquid Haskell [13], refinement types are integrated as special Haskell comments that are interpreted by the refinement type checker and ignored by the Haskell compiler. Thus, refinement types provide a formal verification extension to an existing programming language, preserving the runtime semantics, development tools (e.g., editor and cloud integration support), and optimized libraries of the host language.

Verification is *automated* using SMT solvers [1]. For example, to show that `[1,2,3] !! 2` is safe, the refinement type checker will generate the Verification Condition (VC) `0 ≤ 2 < len [1,2,3]` that is valid only when the code satisfies the refinement type specifications (here safe-indexing). Then, the VC is passed to an SMT solver that will prove it valid, thus the code is safe.

Finally, refinement types are carefully designed to ensure *decidable verification*, which is crucial for both practicality and predictability. To establish decidability, refinement types restrict the language of specifications to a decidable fragment of logic, and design the typing rules, that essentially generate the VCs, so that the logical fragment is preserved. To express more complex concepts, such as quantified properties, verification is type based. For example, `[{v:Int | v ≤ 42}]` expresses the type of lists of integers where *all* elements are smaller than 42, but without the need to quantify over the list elements.

In short, refinement types are designed to be naturally integrated, automated, and decidable, i.e., practical. For that reason, they have already been adopted by various programming languages, e.g., Haskell [12], Ruby [6], Scala [4], and Rust [7]. Yet, in the name of practicality, most refinement type implementations, sacrifice soundness.

A. Ciaffaglione and C. Olarte (Eds.): Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2023)
EPTCS 396, 2023, pp. 1–3, doi:10.4204/EPTCS.396.1

© Niki Vazou
This work is licensed under the
Creative Commons Attribution License.

2 Are Refinement Types Sound?

A verifier is not sound when it accepts a program that violates its specification. An unsoundness error can be generated by two different sources: a bug in the implementation of the verifier or a bug in the design of the refinement type system.

The implementation of refinement type checkers is usually large and error prone. A refinement type checker has to trust the compiler of the underlying language to generate an intermediate program representation (IPR), the type checking rules (adapted to accommodate the IPR) to generate logical verification conditions (VC) and the SMT to validate the VCs. All these three trusted components consist of big code bases that inevitably contain bugs and can potentially lead to unsound verification. Indeed, the implementations of F* [10], Stainless [4], and Liquid Haskell [12] respectively consist of 1.3M, 185.3K, and 423K lines of code, and none of these verifiers isolates a trusted kernel. Approximately five unsoundness bugs per year are reported in each system.

The logic of refinement types is not well understood, leaving it unclear for the users what assumptions are safe to be made and which lead to inconsistencies, and thus unsound verification. For example, we recently discovered [11] that function extensionality had been encoded inconsistently in Liquid Haskell for many years. The inconsistent encoding seemed natural and was assumed by both the developers and users of Liquid Haskell. However under the assumption of functional extensionality Liquid Haskell could prove false, invalidating all the user's verification effort.

Having acknowledged the practicality and unsoundness of refinement types, we argue that the community should focus on making refinement types sound, inspired by techniques from the type theory (e.g., Coq [2], Agda [9], and Lean [8]) that design type systems to be sound by construction.

References

[1] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2017): *The SMT-LIB Standard: Version 2.6*. Technical Report, Department of Computer Science, The University of Iowa. Available at http://www.smt-lib.org/.

[2] Yves Bertot & Pierre Castéran (2004): *Coq'Art: The Calculus of Inductive Constructions*. Springer. Available at https://www.springer.com/gp/book/9783540208549.

[3] Tim Freeman & Frank Pfenning (1991): *Refinement Types for ML*. In: *Programming Language Design and Implementation (PLDI)*, doi:10.1145/113445.113468.

[4] Jad Hamza, Nicolas Voirol & Viktor Kuncak (2019): *System FR: Formalized Foundations for the Stainless Verifier*. In: *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, doi:10.1145/3360592.

[5] Ranjit Jhala & Niki Vazou (2021): *Refinement Types: A Tutorial*. *Foundations and Trends® in Programming Languages*, doi:10.1561/2500000032.

[6] Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster & Emina Torlak (2017): *Refinement Types for Ruby*. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, doi:10.1007/978-3-319-73721-8_13.

[7] Nico Lehmann, Adam T. Geller, Niki Vazou & Ranjit Jhala (2023): *Flux: Liquid Types for Rust*. In: *Programming Language Design and Implementation (PLDI)*, doi:10.1145/3591283.

[8] de Moura L., Kong S., Avigad J., van Doorn F. & von Raumer J. (2015): *The Lean Theorem Prover*. In: *International Conference on Automated Deduction (CADE)*, doi:10.1007/978-3-319-21401-6_26.

[9] Ulf Norell (2009): *Dependently Typed Programming in Agda*. Springer Berlin Heidelberg, doi:10.1007/978-3-642-04652-0_5.

[10] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue & Santiago Zanella-Béguelin (2016): *Dependent Types and Multi-Monadic Effects in F*.* In: *Principles of Programming Languages (POPL)*, doi:10.1145/2837614.2837655.

[11] Niki Vazou & Michael Greenberg (2022): *How to Safely Use Extensionality in Liquid Haskell.* In: *Haskell Symposium*, doi:10.1145/3546189.3549919.

[12] Niki Vazou, Eric L. Seidel & Ranjit Jhala (2014): *LiquidHaskell: Experience with Refinement Types in the Real World.* In: *ACM SIGPLAN Symposium on Haskell (Haskell)*, doi:10.1145/2633357.2633366.

[13] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis & Simon Peyton-Jones (2014): *Refinement Types for Haskell.* In: *International Conference on Functional Programming (ICFP)*, doi:10.1145/2628136.2628161.

Contextual Refinement Types

Antoine Gaulin
McGill University
antoine.gaulin@mail.mcgill.ca

Brigitte Pientka
McGill University
bpientka@cs.mcgill.ca

We develop an extension of the proof environment BELUGA with datasort refinement types and study its impact on mechanized proofs. In particular, we introduce *refinement schemas*, which provide fine-grained classification for the structures of contexts and binders. Refinement schemas are helpful in concisely representing certain proofs that rely on relations between contexts. Our formulation of refinements combines the type checking and sort checking phases into one by viewing typing derivations as outputs of sorting derivations. This allows us to cleanly state and prove the conservativity of our extension.

1 Introduction

Proof mechanization provides strong trust guarantees towards the validity of theorems. Contrary to informal proofs, expressing a theorem and its proof formally requires absolute precision. The resulting statement can thus become riddled with technicalities, which obscures their relation to their informal counterparts. This work combines two approaches to type systems that significantly reduce the added complexity from formalization, namely refinement types and higher-order abstract syntax (HOAS).

Datasort refinement types [10, 9] provide ways to define subtypes (called *datasorts* or just *sorts*) by imposing constraints on the constructors of (inductive) types. Intuitively, a sort S refines a types A if it is defined by a subset of its constructors. The idea originated in the simply-typed setting, where refinements enhance the expressive power of the type system. Later, Lovas and Pfenning [17, 16] extended datasort refinements to the dependently-typed Edinburgh logical framework LF [13]. They provide an equivalence between their system of refinements, LFR, and another extension of LF with proof-irrelevance. An immediate conclusion here is that refinements do not increase the expressive power of dependently-typed calculi. Rather, Lovas observes that refinements may significantly reduce the verbosity of mechanized proofs, which is demonstrated through several case studies [16].

BELUGA is a two-level programming language based on contextual modal type theory (CMTT) [18]. It uses the Edinburgh logical framework LF [13] as a specification logic (data-level), with an intuitionistic first-order reasoning logic (computation-level). The data-level is embedded in the computation-level via a (contextual) box modality similar to the one in the modal logic S4. From a logical point of view, the formula $\Box A$ (read box A) expresses that A is true under no assumptions, i.e. in the empty context. The contextual box modality generalizes this idea to arbitrary contexts, yielding formulas of the form $[\Psi \vdash A]$ expressing that A holds in context Ψ. This allows us to represent LF objects (and types) together with a context in which they are meaningful. To handle this representation, LF contexts are restricted using a notion of *schema* that acts as classifiers of contexts, similarly to how types classify terms. In addition, LF substitutions are first-class objects of BELUGA and they can be used to move objects from one context to another while preserving their meaningfulness. These features allow the expression of an object language (OL) using HOAS [19] and provide several substitution lemmas for free in our mechanizations.

We present BELUGA and its extension with refinement types in Sections 3 and 4, which discuss the data-level and computation-level, respectively. The core of the extension consists of replacing the LF

A. Ciaffaglione and C. Olarte (Eds.): Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2023)
EPTCS 396, 2023, pp. 4–19, doi:10.4204/EPTCS.396.2

© A. Gaulin, B. Pientka
This work is licensed under the Creative Commons Attribution License.

layer of BELUGA with a variation on the LFR system of Lovas and Pfenning [17, 16]. We then lift the refinement relations to the computation-level through straightforward congruence rules and show that the extension is conservative, meaning that every well-sorted program of our extension is well-typed in conventional BELUGA. However, this result only applies if we consider BELUGA as a general-purpose language rather than a proof environment. This is because a BELUGA proof is a recursive function that terminates on every input and refinements allow specifying more precise domains. Thus, the types that we obtain from conservativity can extend the domain of a function, leading to undefined behaviour on certain inputs. In this sense, the extension permits interpreting some partial recursive functions as proofs. While termination is an important part of our work, we focus here on defining the refinements and leave termination checking for future work. More details on this as well as more examples and a full definition of our system will be available in an upcoming technical report [12]. A version of the present paper featuring an appendix containing the fully detailed definitions of our judgments is available on the the first author's website.

2 Motivation

Felty et al. [6] observed that binders can have various structures and that particular results rely only on particular aspects of those structures. This leads to challenges when invoking a lemma in the proof of a theorem since the lemma may rely on simpler binding structures than the theorem. For instance, a lemma using untyped term variables can still be useful for a theorem that uses typed term variables. They propose a series of benchmark challenges along with solutions using multiple contexts and relations between them. We design a new solution based on refinements for one of these benchmarks, namely the equivalence of algorithmic and declarative equalities for the untyped λ-calculus.

The first step of the mechanization is to encode the untyped λ-calculus as an LFR datatype:

```
LFR tm : type =
  | lam : (tm → tm) → tm
  | app : tm → tm → tm;
```

This syntax declares a new type called `tm` whose objects are built from the two given constructors, `lam` and `app`. The constructors encode function abstraction and function application, respectively. Next, we want to encode the judgments for declarative and algorithmic equalities:

```
LFR deq : tm → tm → type =
  | e-lam : ({x : tm} deq x x → deq (M x) (N x)) → deq (lam M) (lam N)
  | e-app : deq M1 N1 → deq M2 N2 → deq (app M1 M2) (app N1 N2)
  | e-refl : {M : tm} deq M M
  | e-sym : deq M N → deq N M
  | e-trans : deq M1 M2 → deq M2 M3 → deq M1 M3;
```

Here, we exploit the dependent types of LFR to express declarative equality as a binary predicate on objects of type `tm`. The constructors `e-refl`, `e-sym`, and `e-trans` encode the axioms of an equivalence relation (reflexivity, symmetry, and transitivity, respectively), while the constructors `e-lam` and `e-app` correspond to congruence rules. Algorithmic equality is just declarative equality without the three equivalence axioms. As such, we define it as a refinement of `deq` rather than as a separate atomic type:

```
LFR aeq ⊏ deq : tm → tm → sort =
  | e-lam : ({x : tm} aeq x x → aeq (M x) (N x)) → aeq (lam M) (lam N)
  | e-app : aeq M1 N1 → aeq M2 N2 → aeq (app M1 M2) (app N1 N2);
```

To declare a new (atomic) sort, users must specify three things: the type which is refined, a refinement kind, and a list of constructors together with their sort. The type must have been previously declared, the sort's kind must refine the type's kind, and each of the constructors' sort must refine their assigned type. By using a sort instead of a type, we get to reuse the same constructors for both judgments. This guarantees that any proof of algorithmic equality can be interpreted as a proof of declarative equality. Thus, we get the soundness of algorithmic equality for free.

The last step in encoding the language is to define its contexts via context schemas. The goal of a schema is to characterize the structure of contexts, which consist of tuples of assumptions rather than being flat lists. A particular context can contain various forms of assumptions (untyped term variables, typed term variables, type variables, etc.) depending on the features of the OL that we are mechanizing. We call these forms of assumptions *worlds* (or *schema elements*). Intuitively, worlds are to schemas what constructors are to atomic types: they specify how to construct a context of a given schema. Our notation for schema declaration emphasizes this idea:

```
schema xdG =
| xeW : block (x : tm, e_x : deq x x);
```

Now, if we have a context Ψ of schema `xdG`, then we can extend it with an additional block variable b with world `xeW`, yielding the context Ψ, b:`xeW`. A refinement of schema is then obtained by selecting a subset of the worlds and refining them to sorts. Here, we want to refine the `deq` assumption to `aeq`, which we do as follows :

```
schema xaG ⊏ xdG =
| xeW : block (x : tm, e_x : aeq x x);
```

One advantage of this approach is that the sort of a block variable `b:xeW` is fully hidden in the schema of the context in which it appears. This means that a given context of schema `xaG` can also be seen as having schema `xdG`. Moreover, this idea generalizes to arbitrary schemas and can go in both directions when all the worlds of the type-level schema also appear in the refinement schema. In this case, given schemas $H \sqsubset G$ and a context $\Psi : H$, we write $\Psi^\top$ to indicate that we wish to interpret Ψ as a context of schema G.

In the conventional solution [8], a relation between contexts of the two schemas needs to be maintained explicitly. Here, we can simply use the refinement relation and our special $\Psi^\top$ context instead. Let us now look at a few cases of the proof of completeness of algorithmic equality:

```
rec aeq-sym : (Ψ : xaG) [Ψ ⊢ aeq M N] → [Ψ ⊢ aeq N M] = ...;
rec ceq : (Ψ : xaG) [Ψ⊤ ⊢ deq M N] → [Ψ ⊢ aeq M N] =
fn d => case d of
  | [Ψ ⊢ #b.2] => [Ψ ⊢ #b.2]
  | [Ψ ⊢ e-sym D] =>
        let [Ψ ⊢ D’] = ceq [Ψ⊤ ⊢ D] in
        aeq-sym [Ψ ⊢ D’]
  | [Ψ ⊢ e-lam (λx.λe. D)] =>
        let [Ψ, b:xeW ⊢ E] = ceq [Ψ, b:xeW ⊢ D[.., b.1, b.2]] in
        [Ψ ⊢ e-lam (λx.λe. E[..,<x;e>])]
  | ...
```

where parentheses around the context variable Ψ : `xaG` indicate implicit quantification.

The first case is for variables, represented as the second projection on one of the block `b` in Ψ. The symbol `#` is merely a syntactic device to identify `b` as a variable. This case acts in essence just like an

identity function, except that the sort of the output does not match that of the input. When we pattern match, we know that `d` has the context $\Psi^\top$ from the sort of `ceq`. Since $\Psi^\top$ has schema `xdG`, we know the world of `b` and therefore that `b.2` has sort `deq b.1 b.1`. On the other hand, when we produce the output `[Ψ ⊢ #b.2]`, then the sort of `ceq` tells us to interpret Ψ as a `xaG`, so we assign it the sort `aeq b.1 b.1`, as desired.

Next, we have the case of symmetry, which is solved by a recursive call on the subderivation, followed by a call to the relevant lemma `aeq-sym`. We know from the sort of `ceq` that the recursive calls produce objects of sort $[\Psi \vdash \texttt{aeq}\ M\ N]$ with $\Psi : \texttt{xaG}$, which is precisely what the lemma expects.

Finally, the case for λ-abstraction requires extending the context with an additional block of assumptions. Here, it is evident that the two contexts involved are the same, except that they are interpreted in different ways.

Our solution is simple and closely resembles an informal proof of completeness of algorithmic equality. In contrast, the conventional BELUGA solution[1] requires a total of 13 additional arguments, including 7 explicit ones that must be manipulated in every case of the proof.

3 Data-level

The main objective of BELUGA is to facilitate reasoning about the properties of OLs. To achieve this, an OL is specified using a variant of the Edinburgh logical framework LF [13], called Contextual LF, in which LF objects and types are always represented together with a context in which they are meaningful, that is containing all of its free variables. The variables of an OL are represented as LF variables, which allows reusing LF's substitution calculus to represent substitution in the OL. This kind of representation is known as HOAS and eliminates the need to prove several substitution properties. Contextual LFR applies the same idea to the LFR system of Lovas and Pfenning [17, 16].

We will start by reviewing Lovas and Pfenning's LFR [17] and discuss the numerous changes that we make to their presentation, and then we will see how refinements carry on to Contextual LFR. Unfortunately, the contextual aspect cannot be cleanly separated from LFR since the syntax and judgments of LFR have to be altered during the extension. In particular, all the judgments depend on an additional context, called the *meta-context* and denoted Ω at the refinement level and Δ at the type level. These consist of *meta-variables* which may occur within LFR objects. So, we maintain these aspects in our presentation of LFR, but defer their explanation to the last part of this section.

We follow a canonical form presentation [28] for the data-level. This means that only normal terms are allowed, which requires the use of hereditary substitutions. Simply put, hereditary substitutions are like ordinary substitutions except that they apply any β-reduction that appears during the process. For instance, the substitution $[(\lambda y.y)/x](x\ 0)$ produces 0 instead of $(\lambda y.y)\ 0$.

3.1 LFR

As previously mentioned, LFR extends LF with datasort refinement types. The objects of LFR are exactly the same as in LF and their classifiers are separated in two levels, types A and sorts S, which are related by a refinement relation $S \sqsubset A$. Due to their dependencies on types, the other syntactic categories are similarly duplicated into a type-level and a sort-level related by a refinement relation.

To facilitate the extension to Contextual LFR, it is crucial that we apply this principle to contexts instead of using a single context containing both typing and sorting assumptions like Lovas and Pfenning

[1] Available at `https://github.com/pientka/ORBI`

[17]. A contextual type $(\Gamma.A)$ encodes the LF judgment that A is a well-formed type in (typing) context Γ and a contextual sort $(\Psi.S)$ similarly encodes the LFR judgment that S is a well-formed sort in (sorting) context Ψ. The only sensible way to establish that $(\Psi.S) \sqsubset (\Gamma.A)$ is to show that $\Psi \sqsubset \Gamma$ and that $S \sqsubset A$. This new refinement relation can then be thought of as a relation between a sort-level judgment and a type-level judgment.

3.1.1 Types and sorts

We start by presenting the types and sorts of LFR and discussing the relations between them. Both types and sorts are allowed to depend on (normal) terms M. They are given by the following syntax:

	Type level	Refinement level
Atomic families	$P ::= \mathbf{a} \mid P\ M$	$Q ::= \mathbf{s} \mid Q\ M \mid P$
Canonical families	$A ::= P \mid \Pi x{:}A_1.A_2$	$S ::= Q \mid \Pi x{:}S_1.S_2$

The refinement relation ultimately boils down to what the user specifies. An atomic type family $\mathbf{a}$ is defined by its constructors and their types. An atomic sort family $\mathbf{s} \sqsubset \mathbf{a}$ is then defined by selecting a subset of the constructors of $\mathbf{a}$ and assigning them sorts that refine their previously specified types. In this sense, refinements offer a way to safely reuse constructors. Finally, the relation is lifted to other types with simple congruence rules:

$$\frac{Q \sqsubset P}{Q\ M \sqsubset P\ M} \qquad\qquad \frac{S_1 \sqsubset A_1 \quad S_2 \sqsubset A_2}{\Pi x{:}S_1.S_2 \sqsubset \Pi x{:}A_1.A_2}$$

Due to the presence of dependencies, type and sort well-formedness are non-trivial in LFR. The type well-formedness judgment, $A : \texttt{type}$, coincides exactly with LF's type well-formedness judgment. It essentially just makes sure that whenever we apply an atomic family P to an argument M, then M has the type prescribed by P's kind. We could likewise consider a sort well-formedness judgment $S : \texttt{sort}$ that validates the sorts of dependencies, but instead we consider the refinement relation itself to be the sort well-formedness judgment. This is also what was done by Lovas and Pfenning [17], however the presence of intersection sorts in their system prevents the full separation of a sort well-formedness judgment. This is because intersections are only allowed when both sorts refine the same type, so the refinement information must be present during the well-formedness derivation.

The refinement relation for atomic families $Q \sqsubset P$ is similar to the notion of constructor subtyping [1], according to which a subtyping relation $P_1 \leq P_2$ occurs between two inductive types when P_1 is defined by a subset of the constructors of P_2. As such, it is sensible to consider a notion of subsorting (i.e. subtyping at the level of sorts) such that $Q \leq P$ whenever $Q \sqsubset P$. In particular, a subsumption rule is admissible for refinements of atomic families. The natural subsorting rule for function spaces would be contra-variant in the domain, while refinement of function spaces is co-variant. As such, a subsumption principle for refinements of function spaces is not guaranteed, although it is admissible for the *weak* function spaces of LF.

In addition to the usual typing judgment $M : A$, we have a sorting judgment, denoted $M :: S$. Sorting replicates typing similarly to how sorts replicate types syntactically. A similar phenomenon occurs for all the other LF judgments (context formation, kinding, etc.). One of our key observations is that the type-level judgments can be unified with their sort-level analogues due to their close resemblances. For typing, this yields a judgment $M : S \sqsubset A$ that encompasses both the facts that $M : A$ and $M :: S$. Let us

exemplify this by considering the rules for function applications:

$$\frac{M_1 : \Pi x{:}A_1.A_2 \quad M_2 : A_1}{M_1\ M_2 : [M_2/x]A_2}\ \text{(Typing)} \qquad \frac{M_1 :: \Pi x{:}S_2.S_2 \quad M_2 :: S_2}{M_1\ M_2 :: [M_2/x]S_2}\ \text{(Sorting)} \qquad \frac{M_1 : \Pi x{:}S_1.S_2 \sqsubset \Pi x{:}A_1.A_2 \quad M_2 : S_1 \sqsubset A_1}{M_1\ M_2 : [M_2/x]S_2 \sqsubset [M_2/x]A_2}\ \text{(Unified)}$$

For the remainder of our presentation, we will focus on this form of unified judgments. In all our judgments, everything that appears to the right of the refinement symbol is considered to be an output. For instance, in $M : S \sqsubset A$, the type A is an output, which amounts to recovering a typing derivation $M : A$ from the sorting derivation $M :: S$. The actual typing rules (see Figure 2) are bi-directional, so we have two unified judgments, one for synthesis and one for checking. This means that we consider neutral terms R and normal terms M, and that we have unified judgments for synthesis ($R \Rightarrow S \sqsubset A$) and checking ($M \Leftarrow S \sqsubset A$). We will discuss this in more details when we introduce terms in 3.1.3.

The other syntactic categories of LFR are similarly duplicated at the refinement level, except for terms which are the same at both levels since they do not contain any type information to refine. Each category is equipped with a refinement relation that is induced by the refinement for types. In all the judgments involving refinements, everything on the right of $\sqsubset$ can be considered as an output of the judgment.

Our presentation differs from that of Lovas and Pfenning [17] in two other ways. First, we use an explicit embedding of types into sorts rather than an ambiguous $\top$ sort that refines every type. This ensures that for any well-formed sort S, we can compute a type A such that $S \sqsubset A$. In turn, this allows us to combine the typing and sorting judgments into a single sorting judgment (see Figure 2). We can then perform type-checking only when it is needed for a sorting derivation, that is when we reach a type embedded into a sort. Moreover, our embedding is at the level of atomic families rather than canonical families. This being said, an embedding of canonical types within canonical sorts is admissible since we can construct a Π-sort from embedded atomic type families. This is also the case for every other syntactic category in BELUGA. Second, we have omitted intersection sorts $S_1 \wedge S_2$, which allow specifying multiple sorts for an object at once.

We note that both subsorting and intersection sorts, although useful features, can significantly slow down sort checking, much like their type-level equivalent would slow down type checking. On the other hand, sorts themselves come at a very low cost while still offering several of the benefits of richer sort systems. The results that we present in 3.1.4 can be extended to a system supporting subsorting and intersection sorts. Adding intersections is straightforward, but subsorting brings complication when it comes to validating coverage since when we pattern match on an object of sort Q, then we need to consider the cases coming from the constructors of Q as usual, but also any additional constructor coming from a subsort of Q.

3.1.2 Contexts and schemas

Next, we take a closer look at LFR contexts and schemas. Again, these are separated into a type-level and a refinement-level that are related by a refinement relation. The syntax of LFR contexts and schemas is as follows:

	Type level	Refinement level
Blocks of declarations	$B ::= \cdot \mid \Sigma x{:}A.B$	$C ::= \cdot \mid \Sigma x{:}S.C$
Schema elements	$E ::= B \mid \Pi x{:}A.E$	$F ::= C \mid \Pi x{:}S.F$
Contexts	$\Gamma ::= \cdot \mid \psi \mid \Gamma, x{:}A \mid \Gamma, b{:}E \cdot \vec{M}$	$\Psi ::= \cdot \mid \psi \mid \Psi, x{:}S \mid \Psi.b{:}F \cdot \vec{M}$
Context schemas	$G ::= \cdot \mid G + E$	$H ::= \cdot \mid H + F$

$\boxed{\Omega \vdash \Psi \sqsubset \Gamma}$ – Refinement relation for contexts

$$\frac{\vdash \Omega \sqsubset \Gamma}{\Omega \vdash \cdot \sqsubset \cdot} \qquad \frac{(\psi : H) \in \Omega}{\Omega \vdash \psi \sqsubset \psi} \qquad \frac{\Omega \vdash \Psi \sqsubset \Gamma \quad \Omega;\Psi \vdash S \sqsubset A}{\Omega \vdash (\Psi, x{:}S) \sqsubset (\Gamma, x{:}A)} \qquad \frac{\Omega \vdash \Psi \sqsubset \Gamma \quad \Omega;\Psi \vdash F \sqsubset E}{\Omega \vdash (\Psi, b{:}F[\vec{M}]) \sqsubset (\Gamma, b{:}E[\vec{M}])}$$

Figure 1: Refinement relations for contexts and schemas

Blocks of declarations represent tuples of labelled assumptions, i.e. variables. The empty block $\cdot$ is not valid on its own, rather it is a syntactic device that indicates the end of a block. Empty blocks are not strictly necessary for the system to work, but facilitate the theoretical development by providing a simple base case.

A schema element is a parameterized block of declarations. While blocks express specific instances of assumptions, schema elements encode the general requirements of a particular form of assumption. For instance, a typing assumption (informally denoted by $x : A$) is characterized by the schema element $\Pi A : \texttt{tp}.\Sigma x{:}\texttt{tm}.\Sigma t{:}\texttt{oft}\ x\ A.\cdot$, while a particular instance of this assumption would be $\Sigma x{:}\texttt{tm}.\Sigma t{:}\texttt{oft}\ x\ \texttt{nat}.\cdot$. Each schema element corresponds to an inference rule for the OL's context formation judgment. A schema is defined as a sum of schema elements and similarly corresponds to the OL's full context formation judgment. Note that in the external syntax used in our example, every schema element was assigned a name and referred to exclusively by that name. Here, we directly use the element's sort to avoid the need for extra premises performing signature lookups in our inference rules.

LFR contexts can contain two kinds of variables. Ordinary variables, denoted by x, stand for an arbitrary LFR object of the specified type. Block variables, denoted by b, stand for tuples of assumptions satisfying the specification of a schema element. In conventional Beluga, block variables are directly assigned with a block of declaration instead of a schema element applied to some objects. Here, we require that these objects be specified explicitly, so that they can be recovered when pattern matching on a context. This simplifies the schema checking rules (see Appendix for a definition) since they no longer rely on unification to establish that a block extension fits a schema element. We also allow a single context variable ψ to appear on the left-most position of LFR contexts Ψ (or Γ), but we do not consider ψ as a variable of Ψ. Instead, ψ is a placeholder for an actual context to be substituted at a later time, so it is stored in the meta-context Ω.

The refinement relations for blocks, schema elements, and schemas are very simple (see Appendix). For schema elements, we just check one sort at a time, starting with the parameters and then the assumptions in the block. Refinements of contexts are similarly checked one assumption at a time (see Figure 1). For block assumptions, we require the same parameters to be used to instantiate the schema elements on both sides of the refinement relation. The relation on schemas is similarly simple, but we take care not to allow duplicate schema elements in G (or in H for that matter). We do this mainly because duplicate elements serve no purpose in practice, but also to highlight the fact that multiple elements of H can refine the same element of G.

Contexts are validated using the schema checking judgments $\Omega \vdash \Psi : H \sqsubset G$ (at the sort-level) and $\Delta \vdash \Gamma : G$ (at the type-level). Assigning a schema H to a context Ψ requires that all the assumptions in Ψ match one of the schema elements in H (see Appendix). This means that all the assumptions in Ψ are of the form $b{:}F[\vec{M}]$, hence there is no rule associated to single variables $x{:}A$. The empty context checks against any well-formed schema. For context extensions, we use an auxiliary judgment $\Omega \vdash \vec{M} : F > D$ that checks the terms in $\vec{M}$ against the parameters of F one at a time. In the end, it produces the block of declarations D obtained by β-reducing $F[\vec{M}]$. Recall that A is an output of the judgment $\Omega; \cdot \vdash M \Leftarrow S \sqsubset A$

and that C is an output of $\Omega \vdash D \sqsubset C$, so we do not need to know them in advance in order to validate the premises.

3.1.3 Terms and substitutions

Now that we have discussed the classifiers of contextual LFR, let us look at the objects that they classify. We distinguish two kinds of objects, namely terms and substitutions. Terms are classified by sorts (and types), while substitutions are classified by contexts. Only normal forms are allowed at the data-level, and this is enforced with a canonical form presentation [28]. The syntax is as follows:

Neutral term	$R ::= \mathbf{c} \mid x \mid b.k \mid R\,M$	n-ary tuple	$\vec{M} ::= \cdot \mid M;\vec{M}$
Normal term	$M ::= R \mid u[\sigma] \mid \lambda x.M$	Substitution	$\sigma ::= \cdot \mid \mathtt{id}_\psi \mid \sigma,M \mid \sigma,\vec{M}$

The separation of terms into neutral and normal ensures that no β-reduction can be done by preventing applications of λ-abstractions. The typing rules (see Figure 2) will also guarantee that all terms are η-long. n-ary tuples of normal terms are crucially used in substitutions to replace block variables b. Since b is always used in a projection $b.k$, the substitution $[\vec{M}/b]$ needs to extract the kth projection of the n-ary tuple in order to avoid expressions of the form $\vec{M}.k$, which are undefined by our grammar (and not normal). This coincides nicely with the idea of hereditary substitution and can be added with only minor modifications to their definition. Substitutions can also contain individual terms, which are used to replace individual variables x. Finally, $\mathtt{id}_\psi$ is the identity substitution for the context variable ψ. Note that substitutions have the same structure as their domain, hence it is not specified explicitly.

$\boxed{\Omega;\Psi \vdash M \Leftarrow S \sqsubset A}$ and $\boxed{\Omega;\Psi \vdash R \Rightarrow S \sqsubset A}$ – Bi-directional typing

$$\frac{(u : \Psi'.P) \in \Omega \quad \Omega;\Psi \vdash \sigma : \Psi' \sqsubset \Gamma}{\Omega;\Psi \vdash u[\sigma] \Leftarrow [\sigma]Q \sqsubset [\sigma]P}$$

$$\frac{(b : F[\vec{M}]) \in \Psi \quad \Omega \vdash \vec{M} : F > C \quad \Omega;\Psi \vdash b : C \gg_1^k S \quad \Omega;\Psi \vdash S \sqsubset A}{\Omega;\Psi \vdash b.k \Rightarrow S \sqsubset A}$$

$\boxed{\Omega;\Psi_1 \vdash \sigma : \Psi_2 \sqsubset \Gamma_2}$ – σ is a well-formed substitution from Ψ_2 to Ψ_1

$$\frac{\Omega;\Psi_1 \vdash \sigma : \Psi_2 \sqsubset \Gamma_2 \quad \Omega;\Psi_2 \vdash \vec{M}_2 : F > D \quad \Omega;\cdot \vdash F \sqsubset E \quad \Omega;\Psi_1 \vdash \vec{M}_1 \Leftarrow D}{\Omega;\Psi_1 \vdash (\sigma,\vec{M}_1) : (\Psi_2, b{:}F[\vec{M}_2]) \sqsubset (\Gamma_2, b{:}E[\vec{M}_2])}$$

Figure 2: Bi-directional typing rules

Neutral terms consist of constants $\mathbf{c}$, single LFR variables x, projections of LFR block variables $b.k$, and function applications of neutral terms to normal terms $R\,M$. The sort synthesis rules for constants, single variables, and function applications are standard and coincide with those of Lovas and Pfenning [17]. Blocks of variables b are not valid LFR objects on their own, instead they are always used in projections. To synthesize the sort of a projection $b.k$, we first retrieve its classifying world $F[\vec{M}]$ from the context, then we compute the block D that it corresponds to via the judgment $\Omega \vdash \vec{M} : F > D$, and finally we extract the kth component of D using the auxiliary judgment $\Omega;\Psi \vdash b : C \gg_i^k$. Normal terms are either neutral terms or λ-abstraction, and the sort-checking rules are standard.

Well-formedness of substitutions is validated with the judgment $\Omega;\Psi_1 \vdash \sigma : \Psi_2 \sqsubset \Gamma$, where Ψ_2 is the domain and Ψ_1 the range of σ. The empty substitution has domain $\cdot$ and range any context Ψ_1, so it allows weakening a closed object to any context. Substitution extension with a single term (σ, M) are validated against context extensions with a single variable $(\Psi_2, x{:}S)$ in the usual way. The substitution $\sigma, \vec{M}$ is used to substitute the n-ary tuple $\vec{M}$ for a block variable b.

3.1.4 Meta-theory

Our main result concerning LFR is that for any refinement-level derivation, there is a corresponding type-level derivation. In particular, well-sorted terms are also well-typed, which means that our extension does not provide any new meaningful terms, i.e. that it is conservative. The statement of the theorem relies on type-level LFR judgments that we have not yet discussed due to their similarities with analogous refinement-level judgments and to the fact we did not change them. Let us then first recapitulate the important refinement-level judgments and mention their type-level analogues:

Judgment	Type-level	Refinement-level
Type formation	$\Delta;\Gamma \vdash A \Leftarrow \mathtt{type}$	$\Omega;\Psi \vdash S \sqsubset A$
Type checking	$\Delta;\Gamma \vdash M \Leftarrow A$	$\Omega;\Psi \vdash M \Leftarrow S \sqsubset A$
Type synthesis	$\Delta;\Gamma \vdash R \Rightarrow A$	$\Omega;\Psi \vdash R \Rightarrow S \sqsubset A$
Schema checking	$\Delta \vdash \Gamma : G$	$\Omega \vdash \Psi : H \sqsubset G$

The same separation occurs for any other judgment of the system. In particular, every refinement relation that we have introduced corresponds to the type-level formation judgment of the associated syntactic category, like for types. The rules defining a type-level judgment are roughly the same as those defining its refinement-level analogue, except that the refinement-level information is replaced by type-level information. Now, we can formulate the conservativity theorem for LFR as follows:

Theorem 3.1.5 (Conservativity for data-level)

1. *If $\Omega;\Psi \vdash S \sqsubset A$, then there are Δ and Γ such that:*
   (a) $\vdash \Omega \sqsubset \Delta$, *(b)* $\Omega \vdash \Psi \sqsubset \Gamma$, *(c)* $\Delta;\Gamma \vdash A \Leftarrow \mathit{type}$.
2. *If $\Omega;\Psi \vdash M \Leftarrow S \sqsubset A$, then there are Δ and Γ such that:*
   (a) $\vdash \Omega \sqsubset \Delta$, *(b)* $\Omega \vdash \Psi \sqsubset \Gamma$, *(c)* $\Delta;\Gamma \vdash M \Leftarrow A$.
3. *If $\Omega \vdash \Psi : H \sqsubset G$, then there are Δ and Ψ such that:*
   (a) $\vdash \Omega \sqsubset \Delta$, *(b)* $\Omega \vdash \Psi \sqsubset \Gamma$, *(c)* $\Delta \vdash \Gamma : G$.

The proof is discussed in 3.2.1. For now, we simply observe that the close resemblance between type-level and refinement-level judgments, combined with the fact that we can extract type-level derivations from refinement-level ones, suggests that we can lift the refinement relations to the level of LFR judgments. This idea will be reinforced by the refinement relations on contextual types.

3.2 Contextual LFR

The contextual layer (also known as the meta-layer [2]) unifies the different kinds of objects and classifiers of the data-level into unique constructs. This facilitates function abstraction at the computation-level since otherwise each kind of object would need a special kind of function space. As before, our classifiers are separated into types and refinement types. Moreover, since contextual objects include LFR contexts, we naturally obtain a refinement relation for objects as well. The syntax is as follows:

	Type level	Refinement level
Contextual types	$\mathcal{A} ::= \Gamma.P \mid \Gamma.\Gamma' \mid G$	$\mathcal{S} ::= \Psi.Q \mid \Psi.\Psi' \mid H$
Contextual objects	$\mathcal{M} ::= \hat{\Gamma}.R \mid \hat{\Gamma}.\sigma \mid \Gamma$	$\mathcal{N} ::= \hat{\Psi}.R \mid \hat{\Psi}.\sigma \mid \Psi$
Meta-contexts	$\Delta ::= \cdot \mid \Delta, X{:}\mathcal{A}$	$\Omega ::= \cdot \mid \Omega, X{:}\mathcal{S}$
Meta-substitutions	$\rho ::= \cdot \mid \rho, \mathcal{M}$	$\theta ::= \cdot \mid \theta, \mathcal{N}$
Meta-variables	$X ::= u \mid \psi$	

Contexts with hats ($\hat{\Gamma}, \hat{\Psi}$) are called *erased* and contain no type or sort information, so they consist only of variables. Erased contexts are sufficient in this setting since the LF neutral objects and LF substitution do not refer to any type or sort information present in the context. Note that if $\Psi \sqsubset \Gamma$, then $\hat{\Psi} = \hat{\Gamma}$. This means that if $\mathcal{N} \sqsubset \mathcal{M}$ are not just contexts, then $\mathcal{N} = \mathcal{M}$. Accordingly, refinements of meta-objects only provides information when contexts are used as objects.

Note that we only allow atomic LFR sorts to occur in the contextual sort $\Psi.Q$ (and similarly at the type-level). This is not a real limitation of the system since the sort $\Psi.\Pi x{:}S_1.S_2$ would be isomorphic to $(\Psi, x{:}S_1).S_2$. Similarly, we allow only neutral terms in contextual objects $\hat{\Psi}.R$, but this does not impact the expressiveness.

A meta-context can contain two kinds of variables, each associated with one of the three possible forms of contextual types. The meta-variable u stands for an LFR term, so it is given the contextual sort $\Psi.Q$. Meta-variables must be associated with an LFR substitution σ before being used in LFR terms as $u[\sigma]$. In this setting, σ is delayed until a meta-substitution is applied to replace u. The other form of meta-variables are context variables ψ, which are assigned a context schema H. Context variables are used to quantify over contexts at the computation-level. The system can also be extended with support for substitution variables [21, 3].

A meta-substitution is similar to an ordinary substitution, except that it substitutes contextual objects for contextual variables. We denote the application of a meta-substitution using double square brackets. For instance, $[\![\theta]\!]M$ applies the meta-substitution θ to the LFR normal term M. A full definition of this operation was given by Cave and Pientka [3].

The refinement relation for contextual types is obtained by lifting the corresponding refinement relations (developed previously). Similarly, the refinement relation for meta-objects is obtained by lifting the refinement relation on LFR contexts. For example, the natural refinement rule for $\Psi.Q$ is the following:

$$\frac{\Omega \vdash \Psi \sqsubset \Gamma \quad \Omega;\Psi \vdash Q \sqsubset P}{\Omega \vdash \Psi.Q \sqsubset \Gamma.P}$$

In a sense, this raises the refinement relation to the level of LFR judgments. It is helpful to pursue this idea further by formulating our rules for the meta- and computation-level as a refinement relation between judgments. The above rule would then become:

$$\frac{\vdash \Omega \sqsubset \Delta \quad \Omega \vdash \Psi \sqsubset \Gamma \quad \Omega;\Psi \vdash Q \sqsubset P}{(\Omega \vdash \Psi.Q) \sqsubset (\Delta \vdash \Gamma.P)}$$

This judgment $(\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A})$ can then serve as both a type well-formedness and a refinement (i.e. sort well-formedness) judgment. We can similarly unify the sorting and typing judgments, the context well-formedness and refinement judgments, and so on (see Appendix). In all of these judgments, the type-level part (everything on the right of $\sqsubset$) can be taken independently for the rest, in which case it defines the usual judgment of conventional BELUGA. On the other hand, we can also consider the type-level part to be an output, which highlights the fact that type-level judgments do not need to be validated prior to their sort-level analogues.

3.2.1 Meta-theory

The new refinement relation between judgments also facilitates formulating our conservativity result since we have all the type information available by assumption:

Theorem 3.2.2 (Conservativity for contextual layer)

1. *If* $(\Omega \vdash \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{A})$, *then* $\Delta \vdash \mathcal{A}$.
2. *If* $(\Omega \vdash \mathcal{N} : \mathcal{S}) \sqsubset (\Delta \vdash \mathcal{M} : \mathcal{A})$, *then* $\Delta \vdash \mathcal{M} : \mathcal{A}$.
3. *If* $(\Omega_1 \vdash \theta : \Omega_2) \sqsubset (\Delta_1 \vdash \rho : \Delta_2)$, *then* $\Delta_1 \vdash \rho : \Delta_2$.

Conservativity for the contextual layer is proven simultaneously with conservativity for LFR due to inter-dependencies between the two. The proof is a straightforward induction on the given derivation.

An important advantage of our formulation of refinements as a relation on judgments is that it eliminates the need for several lemmas, in particular substitution properties. We note that to obtain the full benefits of the approach, we must also formulate the refinements for LF in this style, as otherwise the lemmas are still needed for conservativity of LFR.

4 Computation-level

BELUGA's computation-level is an ML-style functional programming language with support for pattern matching over contextual objects. It features an indexed function space, so that types are allowed to depend only on data-level objects. Contextual objects and types are embedded in the computation-level via a box modality.

4.1 Computation-level refinements

In our extension, the computation-level is separated into a type layer and a refinement layer, just like the data-level. Since contextual objects can occur in computation-level expression, we maintain a refinement relation for expressions in addition to all other syntactic categories. Our presentation is inspired by the one of Pientka and Abel [22], but differs in two important ways. First, we do not consider recursion since it complicates the syntax of patterns significantly. Specifically, valid recursive calls have to be specified as part of every pattern (although they can be inferred, so users do not need to provide them explicitly). Without recursion, patterns are just (boxed) contextual objects. Second, our sorting (and typing) rules do not require coverage for pattern matching. The syntax of the computation-level is the following:

	Type level	Refinement level
Types	τ	$\zeta ::= \; [\mathcal{S}] \mid \zeta_1 \to \zeta_2 \mid \Pi X{:}\mathcal{S}.\zeta$
Contexts	Ξ	$\Phi ::= \; \cdot \mid \Phi, y{:}\zeta$
Expressions	e	$f ::= \; [\mathcal{N}] \mid \texttt{fn}\ y{:}\zeta \Rightarrow e \mid e_1\ e_2 \mid \texttt{mlam}\ X{:}\mathcal{S} \Rightarrow e \mid e\ \mathcal{N}$ $\mid \texttt{let}\ [X] = e_1\ \texttt{in}\ e_2 \mid \texttt{case}^{\zeta}\ [\mathcal{N}]\ \texttt{of}\ \vec{c}$
Branches	b	$c ::= \; \Omega; [\mathcal{N}] \mapsto e$

Meta-types, -sorts, and -objects are embedded into the computation level via a (contextual) box modality, denote $[\mathcal{S}]$ (for sorts). The elimination form for the modality is given by the `let` expressions: an expression $e_1 : [\mathcal{S}]$ is unboxed as the meta-variable X, which may then be used in the expression e_2.

We distinguish two kinds of function spaces, the simple function space $\zeta_1 \to \zeta_2$ and the dependent function space $\Pi X{:}\mathcal{S}.\zeta$. So, dependencies are restricted to objects from the index domain, which provides strong reasoning power over the index domain without all the difficulties of full dependent types.

The language also supports pattern matching on meta-objects through the use of `case` expressions. While we do not allow pattern matching on arbitrary expressions, any expression that has a box sort can be matched against by first unboxing it with a `let` expression and then matching on the new variable. The sort superscript ζ in `case` expression corresponds to the sort invariant that must be satisfied by all the branches in $\vec{c}$. We require that invariants have the form $\Pi\Omega_1.\Pi X_0 : \mathcal{S}_0.\zeta_0$. Intuitively, a branch $\Omega; [\mathcal{N}] \mapsto e$ satisfies the invariant $\Pi\Omega_1.\Pi X_0{:}\mathcal{S}_0.\zeta_0$ if $\mathcal{N}$ has sort $\mathcal{S}_0$ and e has sort $[\![\mathcal{N}/X_0]\!]\zeta_0$, where $\mathcal{N}$, e, and their sorts can depend on Ω.

The judgments for the computation-level have a similar structure as those for contextual LFR. In particular, type-level and refinement-level judgments are performed simultaneously, with the type-level judgment seen as an output of the simultaneous judgment. Since the derivations produced on both sides of the refinement relation are almost exactly the same, we give the rules with only the refinement part. For instance, sorting and typing is expressed as $(\Omega;\Phi \vdash f : \zeta) \sqsubset (\Delta;\Xi \vdash e : \tau)$, but we define only $\Omega;\Phi \vdash f : \zeta$ for conciseness. We focus here on the rules related to pattern matching. The remaining rules are standard and can be found in the appendix. The rule for `case`-expressions is the following:

$$\frac{\zeta = \Pi\Omega_0.\Pi X_0{:}\mathcal{S}_0.\zeta_0 \quad \Omega \vdash \rho : \Omega_0 \quad \Omega \vdash \mathcal{N} : [\![\rho]\!]\mathcal{S}_0 \quad \Omega;\Phi \vdash c : \zeta \ (\text{for all } c \in \vec{c})}{\Omega;\Phi \vdash (\texttt{case}^{\zeta}\ [\mathcal{N}]\ \texttt{of}\ \vec{c}) : [\![\rho,\mathcal{N}/X_0]\!]\zeta_0}$$

The important part of this rule is the last premise, which requires validating that every branch satisfies the given invariant. This is achieved with the judgment $\Omega;\Phi \vdash c : \zeta$ defined by the following rule:

$$\frac{\Omega_0 \vdash \mathcal{N}_0 : \mathcal{S}_0 \quad \Omega,\Omega_0 \vdash \mathcal{S} \doteq \mathcal{S}_0/(\rho,\Omega') \quad \Omega';[\![\rho]\!]\Phi \vdash [\![\rho]\!]f : [\![\rho]\!]\zeta_0}{\Omega;\Phi \vdash (\Omega_0;[\mathcal{N}_0] \mapsto f) : \Pi\Omega_1.\Pi X_0{:}\mathcal{S}_0.\zeta_0}$$

where the judgment $\Omega \vdash \mathcal{S} \doteq \mathcal{S}'/(\rho,\Omega')$ denotes (meta-type) unification. The main difficulty of unification is unifying the dependencies on LF(R) terms. Since we have not modified terms in our extension, the unification algorithm of Pientka and Pfenning [23] still applies.

The conservativity results discussed in section 3.2.1 carry over to the computation-level via straightforward inductions. In particular, every sorting derivation has an analogous typing derivation.

5 Related work

5.1 Refinement types

Various forms of refinement types have been used to solve various problems. Our work is inspired by the datasort tradition that was initiated by Freeman and Pfenning [10, 9] for MINIML, a monomorphic fragment of STANDARD ML's core language. Their system uses refinement type inference so that users do not need to provide annotations, but the inferred sorts are often intersections with some undesired components.

Davies [4], who coined the term datasort, extended this work to the full STANDARD ML language (including modules). They ditched sort inference in favour of a bi-directional sort-checking algorithm, so that only the desired sorts are used by the compiler. Unfortunately, even sort-checking is untenable in the presence of intersection (at least in theory). The compiler needs to choose the correct branch of an intersection when synthesizing sorts for neutral expressions, making sort-checking PSPACE-hard [24].

Jones and Ramsay [15] used refinement types to validate termination of functional programs in the presence of non-exhaustive pattern matching. Their notion of an *intensional* refinement is obtained by removing some of the constructors from a datatype, but the remaining constructors cannot be assigned

new sorts. Instead, a constructor $\mathbf{c} : A$ selected for the sort $\mathbf{s} \sqsubset \mathbf{a}$ is assigned the sort S obtained by replacing every occurrence of $\mathbf{a}$ in A by $\mathbf{s}$. Intensional refinements are weaker than datasorts, but they have full type and refinement inference for a polymorphic ML-style language with algebraic datatype. Their main ideas would also be sufficient to encode our example from section 2.

Another important (and perhaps more common) approach is index refinements, first introduced by Xi and Pfenning [29, 30] for the core language of STANDARD ML. They design a family of dependently-typed ML-style languages parameterized by an arbitrary index domain C, called $\mathrm{DML}(C)$. Refinements are obtained by allowing quantification over the index domain, which intuitively corresponds to having a refinement relation $\Pi x{:}S.A \sqsubset A$, where $S \in C$. In this way, most difficulties of dependent types can be avoided, similarly to how we avoid them in BELUGA's computation-level. Datasort refinements and index refinements were combined by by Dunfield [5], yielding an extension of DML with intersections.

An important development of this approach came in the form of logically qualified (or liquid) types [25], this time as an extension of OCAML. In this methodology, a refinement is expressed as $\{x : \tau \mid P(x)\}$, where τ is a type and P is a boolean-valued function over τ. The type τ can then be seen as the refinement $\{x : \tau \mid \mathtt{true}\}$ and this allows combining typing and sorting into one judgment, much like we have done for datasort refinements.

To our knowledge, there is currently no work on index refinements for dependently-typed languages. A series of papers culminated in the lax logical framework with side conditions $\mathrm{LLF}_{\mathcal{P}}$ [14], which contains types similar in spirit to liquid types. However, $\mathrm{LLF}_{\mathcal{P}}$ uses a notion of *lock* types that is based on monads instead of refinements. This being said, side conditions in $\mathrm{LLF}_{\mathcal{P}}$ can be interpreted as proof irrelevance, which coincides with Lovas and Pfenning's interpretation of sorts in LFR as proof irrelevance [17]. $\mathrm{LLF}_{\mathcal{P}}$ allows more side conditions than LFR, but also puts a heavier proof burden on the user.

5.2 Proof environments

Although HOAS representations offer undeniable benefits, there remains few proof environments that support it. TWELF [20] uses it in its implementation of LF, but all contexts are implicit and this prevents representing context relations [8]. Context schemas emerged from work on TWELF [26], although their notion is more restrictive than ours. The judgments have to be indexed by the unique schema to which the ambient context is known to adhere and there is no way to consider two derivations with different schemas simultaneously.

HYBRID [7] only partially achieves the goals of HOAS: contexts and substitution are inherited from the meta-language, but substitution properties in the OL still need to be proven by hand. However, it has the advantage of being built in well-known logics from which it naturally inherits consistency.

ABELLA [11] does provide the full power of HOAS (called λ-tree syntax in their terminology). Variables are represented in the specification logic through the use of the ∇-quantifier (pronounced nabla). Contexts are represented as lists of ∇-quantified variables and schemas as types depending on these lists. ADELFA [27] is inspired by ABELLA, but uses a specification language that is more closely related to LF. It also improves ABELLA with a built-in notion of schemas.

6 Conclusion

We have developed an extension of Beluga with datasort refinement types. While datasort refinements are mainly used to provide subtyping and intersection types to a language, refinements in the setting

of Beluga offers the potential to express proofs more succinctly. Our extension mainly focused on the notion of refinement schemas, which allow extracting more precise information about contexts, much like refinements extract more precise properties (than types) about objects. In particular, refinement schemas are useful to deal with a special kind of context relations, namely those when the assumptions in two contexts are related by refinements.

6.1 Future work

There are a number of avenues that we plan to explore in the future. First, refinements allow validating the correctness of functions containing non-exhaustive pattern matching thereby supporting modular proof development. A natural next step is therefore to develop a coverage and termination checker for Beluga with refinement types. Due to the close similarity between types and their refinements, it is reasonable to expect that we can adapt [22]. However, challenges emerge from the fact that objects can have multiple sorts (but only one type), especially when it comes to unification. This will in particular impact how we check for coverage.

Second, the refinement relations that we described for schemas and schema elements are limited by the fact that our meta-theory requires sorts to refine only one type. In particular, the relation $D \sqsubset C$ for blocks requires that D and C have the same length. A more interesting relation would allow D to contain more assumptions than C. This is reminiscent of the subtyping principle that adding fields to a record type produces a subtype. Another limitation is that $H \sqsubset G$ can only be established when every element of G is refined by some element of H. Allowing more elements to appear in G would also be meaningful, especially given that every context in G is also in $G + E$ for any element E. With these two modifications, we could represent another class of context relations that Felty et al. [6] calls *linear extensions*. Consequently, our next goal is to provide a more flexible form of refinements and to modify our proof of conservativity so that it no longer relies on type uniqueness for refinements.

References

[1] G. Barthe & M. J. Frade (1999): *Constructor Subtyping*. In S. D. Swierstra, editor: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 109–127, doi:10.1007/3-540-49099-X_8.

[2] A. Cave & B. Pientka (2012): *Programming with binders and indexed data-types*. In J. Field & M. Hicks, editors: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, ACM, pp. 413–424, doi:10.1145/2103656.2103705.

[3] A. Cave & B. Pientka (2013): *First-class substitutions in contextual type theory*. In A. Momigliano, B. Pientka & R. Pollack, editors: *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks & Meta-languages: Theory & Practice, LFMTP 2013, Boston, Massachusetts, USA, September 23, 2013*, ACM, pp. 15–24, doi:10.1145/2503887.2503889.

[4] R. Davies (2005): *Practical Refinement-Type Checking*. Ph.D. thesis, Carnegie Mellon University, USA. AAI3168521.

[5] J. Dunfield (2007): *A Unified System of Type Refinements*. Ph.D. thesis, Carnegie Mellon University. CMU-CS-07-129.

[6] A. P. Felty, A. M. & B. Pientka (2015): *The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 1-A Common Infrastructure for Benchmarks*. *CoRR* abs/1503.06095. arXiv:1503.06095.

[7] A. P. Felty & A. Momigliano (2012): *Hybrid - A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax*. *J. Autom. Reason.* 48(1), pp. 43–105, doi:10.1007/s10817-010-9194-x.

[8] A. P. Felty, A. Momigliano & B. Pientka (2015): *The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations: Part 2-A Survey*. *J. Autom. Reason.* 55(4), pp. 307–372, doi:10.1007/s10817-015-9327-3.

[9] T. S. Freeman (1994): *Refinement Types for ML*. Ph.D. thesis, Carnegie Mellon University, USA, USA. UMI Order No. GAX94-19722.

[10] T. S. Freeman & F. Pfenning (1991): *Refinement Types for ML*. In D. S. Wise, editor: *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, ACM, pp. 268–277, doi:10.1145/113445.113468.

[11] A. Gacek (2008): *The Abella Interactive Theorem Prover (System Description)*. In A. Armando, P. Baumgartner & G. Dowek, editors: *Automated Reasoning*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 154–161, doi:10.1007/978-3-540-71070-7_13.

[12] A. Gaulin (2023): *Contextutual refinement types*. Master's thesis, McGill University. (Forthcoming).

[13] R. Harper, F. Honsell & G. D. Plotkin (1993): *A Framework for Defining Logics*. *J. ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[14] F. Honsell, L. Liquori, P. Maksimovic & I. Scagnetto (2017): $\mathsf{LLF}_{\mathcal{P}}$*: a logical framework for modeling external evidence, side conditions, and proof irrelevance using monads*. *Logical Methods in Computer Science* Volume 13, Issue 3, doi:10.23638/LMCS-13(3:2)2017.

[15] E. Jones & S. Ramsay (2021): *Intensional Datatype Refinement: With Application to Scalable Verification of Pattern-Match Safety*. *Proc. ACM Program. Lang.* 5(POPL), doi:10.1145/3434336.

[16] W. Lovas (2010): *Refinement Types for Logical Frameworks*. Ph.D. thesis, Carnegie Mellon University, USA.

[17] W. Lovas & F. Pfenning (2010): *Refinement Types for Logical Frameworks and Their Interpretation as Proof Irrelevance*. *Log. Methods Comput. Sci.* 6(4), doi:10.2168/LMCS-6(4:5)2010.

[18] A. Nanevski, F. Pfenning & B. Pientka (2008): *Contextual modal type theory*. *ACM Trans. Comput. Log.* 9(3), pp. 23:1–23:49, doi:10.1145/1352582.1352591.

[19] F. Pfenning & C. Elliott (1988): *Higher-Order Abstract Syntax*. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, Association for Computing Machinery, New York, NY, USA, p. 199–208, doi:10.1145/53990.54010.

[20] F. Pfenning & C. Schürmann (1999): *System Description: Twelf — A Meta-Logical Framework for Deductive Systems*. In: *Automated Deduction — CADE-16*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[21] B. Pientka (2008): *A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions*. In G. C. Necula & P. Wadler, editors: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, ACM, pp. 371–382, doi:10.1145/1328438.1328483.

[22] B. Pientka & A. Abel (2015): *Well-Founded Recursion over Contextual Objects*. In T. Altenkirch, editor: *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, *LIPIcs* 38, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 273–287, doi:10.4230/LIPIcs.TLCA.2015.273.

[23] B. Pientka & F. Pfenning (2003): *Optimizing Higher-Order Pattern Unification*. In F. Baader, editor: *Automated Deduction – CADE-19*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 473–487, doi:10.1007/978-3-540-45085-6_40.

[24] J. C. Reynolds (1997): *Design of the Programming Language Forsythe*, pp. 173–233. Birkhäuser Boston, Boston, MA, doi:10.1007/978-1-4612-4118-8_9.

[25] P.M. Rondon, M. Kawaguci & R. Jhala (2008): *Liquid Types*. *SIGPLAN Not.* 43(6), p. 159–169, doi:10.1145/1379022.1375602.

[26] C. E. Schürmann (2000): *Automating the Meta Theory of Deductive Systems*. Ph.D. thesis, Carnegie Mellon University, USA. AAI9986626.

[27] M. Southern & G. Nadathur (2021): *Adelfa: A System for Reasoning about LF Specifications*. In E. Pimentel & E. Tassi, editors: *Proceedings of the Sixteenth Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2021, Pittsburgh, USA, 16th July 2021*, *EPTCS* 337, pp. 104–120, doi:10.4204/EPTCS.337.8.

[28] K. Watkins, I. Cervesato, F. Pfenning & D. Walker (2002): *A concurrent logical framework I: Judgments and properties*. Technical Report CMU-CS-02-101, Carnegie Mellon University.

[29] H. Xi (1998): *Dependent Types in Practical Programming*. Ph.D. thesis, Carnegie Mellon University, USA. AAI9918624.

[30] H. Xi & F. Pfenning (1999): *Dependent Types in Practical Programming*. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, Association for Computing Machinery, New York, NY, USA, p. 214–227, doi:10.1145/292540.292560.

Semi-Automation of Meta-Theoretic Proofs in Beluga

Johanna Schwartzentruber
McGill University
Montreal, Canada
johanna.schwartzentruber@mail.mcgill.ca

Brigitte Pientka
McGill University
Montreal, Canada
bpientka@cs.mcgill.ca

We present a sound and complete focusing calculus for the core of the logic behind the proof assistant Beluga as well as an overview of its implementation as a tactic in Beluga's interactive proof environment Harpoon. The focusing calculus is designed to construct uniform proofs over contextual LF and its meta-logic in Beluga: a dependently-typed first-order logic with recursive definitions. The implemented tactic is intended to complete straightforward sub-cases in proofs allowing users to focus only on the interesting aspects of their proofs, leaving tedious simple cases to Beluga's theorem prover. We demonstrate the effectiveness of our work by using the tactic to simplify proving weak-head normalization for the simply-typed lambda-calculus.

1 Introduction

To establish trust in a software system, we need to begin with establishing trust in the programming language (PL) that is used in its implementation. To model formal systems like PLs, we first need to specify them in a specification logic, and then prove statements about the behaviour of the specified system within another logic, called the reasoning logic. A key challenge in specifying languages is how to model variable bindings, variable renaming, α-conversion, substitution, and reasoning under assumptions. Higher-order abstract syntax (HOAS) offers a solution to these issues, reducing the amount of overhead infrastructure users must construct. On the reasoning side, proofs about these language specifications for even small languages can still involve long and tedious proofs where many sub-cases are trivial, and the only challenging aspect pertains to one or two interesting parts. Our goal is to ease the burden on specifying and proving properties by providing users with automated support based on proof theoretic foundations to discharge trivial, straightforward cases automatically.

There are several HOAS-based systems that offer varying degrees of automation. Abella [14] allows users to develop proofs interactively using a small set of tactics which offer little automation. Hybrid [2, 18] adds HOAS support to the general proof assistants Coq [6] and Isabelle/HOL [20] and consequently is able to harness their automation and tooling powers, at the cost of added work to specify systems [13, 12]. Twelf offers full automation; although users are not able to interact with its prover which does not support backtracking. Most importantly, Twelf and Abella do not produce proof terms and therefore offer no way to verify their proofs independently.

We investigate proof automation within the HOAS-based proof assistant, Beluga [25]. In Beluga, users formalize their systems within the logical framework LF [15] and subsequently prove properties about LF objects in Beluga's reasoning logic: a dependently typed first-order logic. Beluga takes a functional approach; modelling inductive proofs about LF objects

A. Ciaffaglione and C. Olarte (Eds.): Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2023)
EPTCS 396, 2023, pp. 20–35, doi:10.4204/EPTCS.396.3

© J. Schwartzentruber & B. Pientka
This work is licensed under the Creative Commons Attribution License.

as recursive dependently-typed programs, following the Curry-Howard isomorphism. To ease proof developments, a tactic-based prover HARPOON was recently added [11]. Users may now construct proofs interactively using a small set of high-level actions similar to those in Abella, and, upon completion of a proof, will be presented with a proof script which can be translated to a BELUGA program that may be independently type-checked. It is still the case however that much human interaction is required for HARPOON proof developments.

In this paper we present a focusing calculus designed to build uniform proofs over the core of BELUGA's two-level logic, and provide an overview of its implementation as a HARPOON tactic. The tactic is meant to solve simple lemmas and simple cases of proofs, allowing users to concentrate on the interesting aspects of a proof. We have proven that the focusing calculus, presented in Section 2.3, is sound and complete with respect to the cut-free sequent calculus for BELUGA, presented in Section 2.2.2. We have used our tactic on a number of interesting case studies in PL theory, which we summarize in Section 3.2, including type preservation and value soundness for MiniML (without fix points), weak-head normalization for the simply-typed lambda-calculus, and the Church-Rosser theorem for the untyped lambda-calculus. We have seen that they allow for automatic completion of many of the simpler lemmas and subcases of these theorems. We believe automating proof search over the core of BELUGA results in simpler proof developments, making it more appealing to users looking to verify formal systems.

2 Introduction to Beluga

In this section, we present an overview of the BELUGA system. We begin with an informal, followed by a formal, description of its logic, concluding with a presentation of the focusing calculus that we implement.

2.1 Encoding (meta-)theories

We introduce BELUGA informally by demonstrating how theories and their meta-theories are encoded within the system. Throughout this paper we will focus on the proof development of a key lemma that is used to prove weak-head normalization for the simply-typed lambda-calculus using logical relations. The lemma states that reducibility is closed under expansion, i.e. if term `M` steps to term `N` and `N` reduces to type `A`, then `M` does as well. A detailed description of the full mechanization may be found at [9].

We begin by encoding the theory of the simply-typed lambda-calculus within BELUGA's specification logic, the logical framework LF [15] making use of HOAS. We choose an intrinsically-typed representation to simplify our mechanization.

```
LF tp : type =
| b : tp
| arr : tp → tp → tp ;
```

```
LF term : tp → type =
| app : term (arr A B) → term A → term B
| abs : tp → (term A → term B) → term (arr A B)
| c : term b ;
```

Next, we define the operational semantics of our lambda terms. For simplicity we do not reduce under abstractions. We observe that object-level substitution is modelled by LF application, as in the type of `beta`.

```
LF step : term A → term A → type =
| beta : step (app (abs A M) N) (M N)
| stepapp : step M M'
            → step (app M N) (app M' N) ;
```

```
LF steps : term A → term A → type =
| id   : steps M M
| sstep : step M M'
            → steps M' M'' → steps M M'' ;
```

To complete the theory's encoding, we also define what it means for a term to halt, i.e. it steps to a value.

```
LF val : term A → type =
| val/c : val c
| val/abs : val (abs A M) ;
```

```
LF halts : term A → type =
| halts/m : steps M M' → val M' → halts M ;
```

To reason about LF objects, we embed them within the computation logic using a modal box (necessity) operator [23]. Users also have the ability to reason about "open" LF objects in Beluga. To do this, we pair each LF object together with the LF context in which it is meaningful [23, 24]. This concept is internalized as a contextual type $\lceil \Psi \vdash P \rceil$ [19]. This contextual type describes a computation-level expression box $(\widehat{\Psi} \vdash R)$ where R is an LF object of type P in the context Ψ. In other words, $\widehat{\Psi}$ describes the free variables in R and corresponds to the erased typing context Ψ.

We present a trivial result about our specified theory to demonstrate how meta-theorems are expressed in Beluga.

```
rec halts_step : [ ⊢ step M M'] → [ ⊢ halts M'] → [ ⊢ halts M] =
fn s, h =>
   let [ ⊢ halts/m MS V] = h in let [ ⊢ S] = s in [ ⊢ halts/m (sstep S MS) V] ;
```

We leave the contextual variables `M`, `M'`, and `M''` implicitly universally quantified as Beluga is able to reconstruct their type. We use Beluga's simple function space to formalize our implication statement. Recall proofs are programs in Beluga, therefore proof development proceeds in a functional manner.

We begin by giving the theorem name and statement, prefixed with the keyword **`rec`**. The proof starts by peeling off the implication antecedents (**`fn`** `s, h =>`). Working backwards, we know we must use `halts/m` to construct our desired term as it is currently the only constructor for terms of type `halts`, therefore we must solve its subgoals, namely that there is a value that `M` steps to. We first invert `h` as it has only one possible constructor. This reveals that it is actually the contextual object `[ ⊢ halts/m MS V]` where `MS` and `V` are LF terms of type `steps M' N` and `val N` (for some implicit meta-variable `N`) respectively. It may appear we have every piece of the puzzle required to solve our goal: we have a value that our term `M` steps too. However once we transition to the LF level to build our LF proof term, we do not have access to our computation-level context, in which `s` resides. Therefore, we must first *unbox* said assumption.

The free variables appearing in all the specifications above are treated as implicitly quantified. Beluga infers their types during type reconstruction. As such, users do not supply arguments for such parameters.

2.2 Theoretical foundation

We give a formal presentation of the core of Beluga's two-level logic beginning with its grammar followed by its two-level proof system described using two cut-free sequent calculi. For a full description of the logic, readers may refer to past works [8, 23].

2.2.1 Grammar

BELUGA's two-level logic is based on dependent contextual modal type theory (CMTT) [19]. At the core of the specification logic is the logical framework LF [15] which supports encodings using HOAS. We give here a definition that characterizes only canonical (normal) objects as these are the only ones that are meaningful in our setting.

We separate terms into two categories, neutral and normal. We characterize neutral terms to be those that do not cause beta-redexes when they are applied in function application. Terms are classified by types, which are either type constants **a** that may be indexed by terms $M_1, \ldots, M_n$, or dependent types.

Atomic Types	P, Q	$::=$	$\mathbf{a}\overrightarrow{M}$	Substitutions	σ	$::=$	$\cdot \mid \mathsf{wk}_\psi \mid \sigma, M$
Types	A, B	$::=$	$P \mid \Pi x{:}A.\, B$	Contexts	Ψ, Φ	$::=$	$\cdot \mid \Psi, x : A \mid \psi$
Neutral Terms	R	$::=$	$x \mid \mathbf{c} \mid R\ N \mid u[\sigma]$	Contextual Variables	X	$::=$	$u[\sigma] \mid \psi$
Normal Terms	M, N	$::=$	$R \mid \lambda x.\, M$				

To support pattern matching on LF objects, we further extend LF with two kinds of contextual variables: meta-variables, written as $u[\sigma]$ and context variables, written as ψ. Context variables allow for abstraction over contexts which is required for recursion over HOAS specifications. Meta-variables allow us to describe "holes" in an LF object. They describe possibly open objects that are paired with a postponed simultaneous substitution σ (by convention written to the right of a term) that is applied as soon as we know what u stands for.

Simultaneous substitutions σ provide a mapping from one context of variables Φ to another Ψ. We do not always make the domain of the substitution explicit, but one can think of the i-th element of σ corresponding to the i-th declaration in Φ. We assume all substitutions are hereditary substitutions [33].

Variables in a contextual LF expression may be bound by one of two contexts. There is the LF context Ψ that holds typings for ordinary variables, and there is the meta-context Δ (introduced below) which holds typings for contextual variables, uniformly denoted by X. Contextual variables include meta-variables and context variables ψ.

In order to uniformly abstract over meta-objects in the computation logic, we lift contextual LF objects to meta-types U and meta-terms C.

Meta Terms	C	$::=$	$(\hat{\Psi} \vdash R) \mid \Psi$	Context Schemas	G	$::=$	$\exists\overrightarrow{(x{:}A_o)}.\ A \mid G + \exists\overrightarrow{(x{:}A_o)}.\ A$
Meta Types	U	$::=$	$(\Psi \vdash P) \mid G$	Meta Contexts	Δ	$::=$	$\cdot \mid \Delta, X : U$
Meta Substitutions	θ	$::=$	$\cdot \mid \theta, C/X$				

The core of our meta language's terms include contextual terms as well as LF contexts. The meta-type $(\Psi \vdash P)$ denotes the type of a meta-variable u and stands for a contextual term. For simplicity, we restrict context schemas G to be constructed from schema elements $\exists\overrightarrow{(x{:}A_o)}.A$ using $+$, where A is an LF type.

We write the single meta-substitution as $[\![C/X]\!]$. In most cases X stands for a meta-variable u and C stands for a contextual object $(\hat{\Psi} \vdash R)$. In this case the substitution gets pushed through λ-expressions until we reach a meta-variable $u[\sigma]$. We then apply the meta-substitution to its associated substitution to obtain σ' before eagerly applying σ' to R. The full definition of meta-substitutions may be found at [8, 23].

On top of contextual LF we have the computational layer, which is used to describe programs that operate on data. The computation types include atomic box-types $\lceil \Psi \vdash P \rceil$, computation level function abstraction, as well as abstraction over contextual objects.

Types	τ	::=	$\lceil \Psi \vdash P \rceil \mid \tau_1 \to \tau_2 \mid \Pi^{\square} X{:}U.\,\tau$
Expressions	E	::=	$y \mid \mathsf{box}\ (\hat{\Psi} \vdash R) \mid \mathsf{let\ box}\ X = E_1\ \mathsf{in}\ E_2$
			$\mid \mathsf{fn}\ y.E \mid E_1\ E_2 \mid \lambda^{\square} X.E \mid E\ (C)$
Computation-level Contexts	Γ	::=	$\cdot \mid \Gamma, y : \tau$

Ordinary functions are created using $\mathsf{fn}\ y.E$, while we write $\lambda^{\square}X.E$ for dependent functions that abstract over meta-objects. We overload the application operation. We write $E_1\ E_2$ for applying the expression E_1 of function type $\tau_1 \to \tau_2$ to an expression E_2. We also write $E\ (C)$ for applying the expression E of type $\Pi^{\square}X{:}U.\,\tau$ to a contextual object C.

2.2.2 Sequent calculi

We now present the sequent calculus for contextual LF, which is based on the sequent calculus for intuitionistic contextual modal logic presented in [19]. Following the logic programming interpretation of LF, a proof of a proposition encoded as an LF type is an LF term which inhabits said type. We exploit the fact that if B does not depend on x in $\Pi x{:}A.B$, we interpret it as an implication $A \to B$. If x does occur in B, then we treat $\Pi x{:}A.B$ as universal quantification, written as $\Pi \hat{x}{:}A.B$.

All sequents have access to the global signature Σ which we keep implicit. The sequent $\Delta; \Psi \Longrightarrow M : A$ states that M is a proof of the proposition A, or M has type A, using assumptions from the meta-context Δ and the LF context Ψ. As a consequence, we keep in the LF context Ψ not only proof-relevant assumptions that arise from implications (in particular $\to R$), but we also add parameter assumptions that come from ΠR. We distinguish between these two assumptions by writing $x{:}A$ for the former, and $\hat{x} : A$ for the latter. This is emphasized in the init^{Ψ} rule, where only proof-relevant assumptions are used to finish proofs.

In addition to the sequent $\Delta; \Psi \Longrightarrow M : A$, we also have $\Delta; \Psi \Longrightarrow \sigma : \Phi$ which states that the substitution σ witnesses a proof of the propositions in Φ using assumptions from Δ and Ψ. Intuitively, Φ is an LF context containing assumptions $x_i{:}A_i$ and $\hat{x}_j{:}A_j$. Note that we need to construct a proof for the former, but for the latter we are able to determine the witness via unification in practice.

The rules for the sequent calculus for contextual LF are presented in Figure 1, on the next page. The right rules introduce variable declarations into the local context Ψ. However, those introduced via the ΠR rule are simply parameters that are not used during proof search, unlike those introduced via $\to R$. To use a universally quantified assumption (i.e. a dependent function type) as in ΠL we require that M checks against type A, written $\Delta; \Psi \vdash M \Leftarrow A$, in the appropriate contexts. In practice, we do not search for the term M but introduce meta-variables for such universally quantified variables which are later instantiated via unification. In contrast, using an assumption of ordinary function type, as in $\to L$, involves searching for a proof term of type A. Note that in the left rules we replace a neutral term by a neutral term, thus still constructing normal proofs.

In the reflect rule we may use the contextual assumption $(\Phi \vdash P)$ to deduce P in the context Ψ if we can verify Φ. In order to verify Φ we need to find a substitution which maps all the variables in Φ to terms that make sense in Ψ. There are several ways to construct such a substitution, depending on the shape of Φ. If it is empty, we simply use an empty substitution (as P is closed). If it is a context variable ψ and we simply want to use $(\psi \vdash P)$ in a weaker context, we apply the weakening substitution. Otherwise, Φ contains two different kinds of

$\boxed{\Delta;\Psi \Longrightarrow M : A}$ Object M is a proof term for the proof of A in the sequent calculus

$$\frac{\mathbf{c} : A \in \Sigma}{\Delta;\Psi \Longrightarrow \mathbf{c} : A}\ \text{init}^{\Sigma} \qquad \frac{}{\Delta;\Psi, x : A \Longrightarrow x : A}\ \text{init}^{\Psi} \qquad \frac{\Delta;\Psi,\hat{x} : A \Longrightarrow M : B}{\Delta;\Psi \Longrightarrow \lambda x.M : \Pi\hat{x}{:}A.\, B}\ \Pi R$$

$$\frac{\Delta;\Psi, x_1 : \Pi\hat{x}{:}A.\ B \vdash M \Leftarrow A \quad \Delta;\Psi, x_1 : \Pi\hat{x}{:}A.\, B, x_2 : [M/\hat{x}]B \Longrightarrow N : A'}{\Delta;\Psi, x_1 : \Pi\hat{x}{:}A.\, B \Longrightarrow [x_1 M/x_2]N : A'}\ \Pi L$$

$$\frac{\Delta;\Psi, x : A \Longrightarrow M : B}{\Delta;\Psi \Longrightarrow \lambda x.M : A \to B} \to R \qquad \frac{\Delta;\Psi, x_1 : A \to B \Longrightarrow M : A \quad \Delta;\Psi, x_1 : A \to B, x_2 : B \Longrightarrow N : A'}{\Delta;\Psi, x_1 : A \to B \Longrightarrow [x_1 M/x_2]N : A'} \to L$$

$$\frac{\Delta, u : (\Phi \vdash P);\Psi \Longrightarrow \sigma : \Phi \quad \Delta, u : (\Phi \vdash P);\Psi, x : [\sigma]P \Longrightarrow M : A}{\Delta, u : (\Phi \vdash P);\Psi \Longrightarrow [u[\sigma]/x]M : A}\ \text{reflect}$$

$\boxed{\Delta;\Psi \Longrightarrow \sigma : \Phi}$ Object σ is a substitution that witnesses the proof of Φ in the sequent calculus

$$\frac{}{\Delta;\Psi \Longrightarrow \cdot : \cdot}\ \text{sub}_{\text{empty}} \qquad \frac{}{\Delta;\psi,\Psi \Longrightarrow \mathsf{wk}_{\psi} : \psi}\ \text{sub}_{\mathsf{wk}}$$

$$\frac{\Delta;\Psi \Longrightarrow \sigma : \Phi \quad \Delta;\Psi \Longrightarrow N : [\sigma]B}{\Delta;\Psi \Longrightarrow (\sigma, N) : (\Phi, x{:}B)}\ \text{sub}_p \qquad \frac{\Delta;\Psi \Longrightarrow \sigma : \Phi \quad \Delta;\Psi \vdash M \Leftarrow [\sigma]B}{\Delta;\Psi \Longrightarrow (\sigma, M) : (\Phi, \hat{x}{:}B)}\ \text{sub}_u$$

Figure 1: Sequent calculus for contextual LF.

variable declarations: a declaration $x{:}B$ requires proof search in order to find a term N in Ψ of type $[\sigma]B$ to replace x in P; a declaration $\hat{x}{:}A$ stands for a universally quantified variable and does not require proof search. In practice, we can determine it via unification. This different treatment of assumptions reflects their different roles.

We turn our attention to proof search over computations (see Figure 2). Our inference rules are mostly standard for a first-order logic. The $\Box R$ rule is the transition rule between contextual LF and computation-level proofs. In $\Box L$ we *unbox* a boxed assumption, adding it to Δ. Using computation assumptions as in $\Pi^{\Box} L$ and $\to L$ is similar to how contextual LF assumptions are used. To use a universally quantified assumption, as in $\Pi^{\Box} L$, we require that C checks against U. Again, this term C is not explicitly constructed but found instead through unification. We note that meta-objects may only depend on the meta-context, hence we only require that C check against type U in Δ.

We show in [31] that cut and contextual cut are admissible in the computation logic. These results are shown for contextual LF in [19]. Using the admissibility of cut results, we can deduce invertibility of some of the rules in our calculi. Interestingly, the box constructor $\Box$ has an invertible *left* rule, which will have implications in the focusing calculus. We omit the proof terms for readability here.

Lemma 1 (Invertibility in the sequent calculi)**.**
a) (ΠR) If $\Delta;\Psi \Longrightarrow \Pi\hat{x}{:}A.B$ *then* $\Delta;\Psi,\hat{x} : A \Longrightarrow B$
b) ($\to R$) If $\Delta;\Psi \Longrightarrow A \to B$ *then* $\Delta;\Psi, x : A \Longrightarrow B$
c) ($\Pi^{\Box} R$) If $\Delta;\Gamma \Longrightarrow \Pi^{\Box} X{:}U.\tau$ *then* $\Delta, X : U;\Gamma \Longrightarrow \tau$
d) ($\to R$) If $\Delta;\Gamma \Longrightarrow \tau_1 \to \tau_2$ *then* $\Delta;\Gamma, y : \tau_1 \Longrightarrow \tau_2$
e) ($\Box L$) If $\Delta;\Gamma, y : \lceil \Psi \vdash P \rceil \Longrightarrow \tau$ *then* $\Delta, X : (\Psi \vdash P);\Gamma, y : \lceil \Psi \vdash P \rceil \Longrightarrow \tau$

$\boxed{\Delta;\Gamma \Longrightarrow E:\tau}$ Object E is a proof term for the proof of τ in the sequent calculus

$$\frac{\Delta, X:U;\Gamma \Longrightarrow E:\tau}{\Delta;\Gamma \Longrightarrow \lambda^{\Box}X.E:\Pi^{\Box}X{:}U.\tau}\ \Pi^{\Box}R \qquad \frac{\Delta \Vdash C \Leftarrow U \quad \Delta;\Gamma, y_1:\Pi^{\Box}X{:}U.\tau', y_2:[\![C/X]\!]\tau' \Longrightarrow E:\tau}{\Delta;\Gamma, y_1:\Pi^{\Box}X{:}U.\tau' \Longrightarrow [y_1\ (C)/y_2]E:\tau}\ \Pi^{\Box}L$$

$$\frac{\Delta;\Gamma, y_1:\tau_1\to\tau_2 \Longrightarrow E':\tau_1 \quad \Delta;\Gamma, y_1:\tau_1\to\tau_2, y_2:\tau_2 \Longrightarrow E:\tau}{\Delta;\Gamma, y_1:\tau_1\to\tau_2 \Longrightarrow [y_1\ E'/y_2]E:\tau}\ \to L$$

$$\frac{\Delta;\Gamma, y:\tau_1 \Longrightarrow E:\tau_2}{\Delta;\Gamma \Longrightarrow \mathsf{fn}\ y.\ E:\tau_1\to\tau_2}\ \to R \qquad \frac{}{\Delta;\Gamma, y:\tau \Longrightarrow y:\tau}\ \text{init} \qquad \frac{\Delta;\Psi \Longrightarrow R:P}{\Delta;\Gamma \Longrightarrow \mathsf{box}\ (\hat{\Psi}\vdash R):\lceil\Psi\vdash P\rceil}\ \Box R$$

$$\frac{\Delta, X{:}(\Psi\vdash P);\Gamma, y{:}\lceil\Psi\vdash P\rceil \Longrightarrow E:\tau}{\Delta;\Gamma, y{:}\lceil\Psi\vdash P\rceil \Longrightarrow \mathsf{let\ box}\ X = y\ \mathsf{in}\ E:\tau}\ \Box L$$

Figure 2: Sequent calculus for the computation logic.

2.3 Focused proof system

As a foundation for automating proof search in Beluga we develop a focused sequent calculus over Beluga's two-level logic. For the description of this focused proof system, we omit proof terms when permitted to ease readability. This logic formalizes the proof search procedure that we implement. The loop is fully automatic and therefore requires that non-determinism be handled with ease. The sequent calculus previously presented does not suffice as the rules do not provide any inherent direction for proof search. The rules of the following calculi guide better proof development. The calculus builds uniform proofs [17] by applying all invertible rules first. We then handle non-invertible rules systematically through focusing [3].

The focusing calculus for contextual LF consists of two main phases: a uniform and focusing phase, and is mostly straightforward (see Figure 3). The uniform proof phase consists of applying the invertible rules until we reach an atomic goal (P). During this phase, parameters and assumptions are collected and placed in the LF context. We then try to find a solution by focusing on assumptions from the different contexts. In particular, we iterate through assumptions (in the meta and LF contexts), decomposing each one into the atoms it defines without utilizing any other assumption.

In the transition$^{\Delta}$ rule, using an assumption $(\Phi \vdash Q)$ from Δ to complete a proof requires a simultaneous substitution (σ) to be constructed so that Q makes sense in the current LF context Ψ. We find such a substitution through proof search. When focusing on assumptions from Ψ of function type, we search for a proof of A if our assumption is of non-dependent function type. Otherwise the assumption is of dependent function type, in which case M is found via unification.

Proof search over the reasoning layer proceeds similarly to LF (see Figure 4). We first perform all invertible rules, then we must make a choice on what to focus on. Unlike in LF proof search, proof search over computations requires two separate phases of inversions since the box connective has an invertible left rule. Further, in addition to focusing on the left, we also focus on the right, which corresponds to proof search in LF.

We begin with a uniform right phase which ends with an atomic goal formula, $\lceil\Psi\vdash P\rceil$. From

$\boxed{\Delta; \Psi \overset{u}{\Longrightarrow} A}$ There is a uniform proof of A in the focusing calculus

$$\frac{\Delta; \Psi, \hat{x}{:}A \overset{u}{\Longrightarrow} B}{\Delta; \Psi \overset{u}{\Longrightarrow} \Pi\hat{x}{:}A.\ B}\ \Pi R \qquad \frac{\Delta; \Psi, x : A \overset{u}{\Longrightarrow} B}{\Delta; \Psi \overset{u}{\Longrightarrow} A \to B}\ \to R$$

$$\frac{\Delta(X) = (\Phi \vdash Q) \quad \Delta; \Psi \overset{u}{\Longrightarrow} \sigma : \Phi \quad [\sigma]Q = P}{\Delta; \Psi \overset{u}{\Longrightarrow} P}\ \text{transition}^{\Delta} \qquad \frac{\Psi(x) = A \quad \Delta; \Psi > x : A \rightrightarrows P}{\Delta; \Psi \overset{u}{\Longrightarrow} P}\ \text{transition}^{\Psi}$$

$\boxed{\Delta; \Psi \overset{u}{\Longrightarrow} \sigma : \Phi}$ Object σ is a substitution that witnesses a uniform proof of Φ in the focusing calculus

$$\frac{}{\Delta; \Psi \overset{u}{\Longrightarrow} \cdot : \cdot}\ \text{empty} \qquad \frac{}{\Delta; \psi, \Psi \overset{u}{\Longrightarrow} \mathsf{wk}_{\psi} : \psi}\ \text{sub}_{\mathsf{wk}}$$

$$\frac{\Delta; \Psi \overset{u}{\Longrightarrow} \sigma : \Phi \quad \Delta; \Psi \overset{u}{\Longrightarrow} N : [\sigma]B}{\Delta; \Psi \overset{u}{\Longrightarrow} (\sigma, N) : (\Phi, x{:}B)}\ \text{sub}_p \qquad \frac{\Delta; \Psi \overset{u}{\Longrightarrow} \sigma : \Phi \quad \Delta; \Psi \vdash M \Leftarrow [\sigma]B}{\Delta; \Psi \overset{u}{\Longrightarrow} (\sigma, M) : (\Phi, \hat{x}{:}B)}\ \text{sub}_u$$

$\boxed{\Delta; \Psi > x : A \rightrightarrows P}$ There is a focused proof of P with focus on x in the focusing calculus

$$\frac{}{\Delta; \Psi > x : P \rightrightarrows P}\ \text{init}^{\Psi} \qquad \frac{\Delta; \Psi \overset{u}{\Longrightarrow} A \quad \Delta; \Psi > x' : B \rightrightarrows P}{\Delta; \Psi > x : A \to B \rightrightarrows P}\ \to L$$

$$\frac{\Delta; \Psi \vdash M \Leftarrow A \quad \Delta; \Psi > x' : [M/\hat{x}]B \rightrightarrows P}{\Delta; \Psi > x : \Pi\hat{x}{:}A.\ B \rightrightarrows P}\ \Pi L$$

Figure 3: Focusing calculus for contextual LF.

there we transition to the uniform left phase where we unbox all box-type assumptions in Γ, moving them to Δ. This is done because when we shift levels to LF proof search we only bring with us assumptions that are true across all levels (those in Δ) and computation assumptions do not make sense on the LF level. The sequent for the uniform left phase is novel. We use the symbol $\gg$ as a way to distinguish assumptions that may be of box-type (to the right of $\gg$) from ones that are not (to the left of $\gg$). Recall that the order of assumptions in Γ does not matter, therefore it is acceptable that the order reverses each time we complete a uniform left phase.

Similarly to focusing in LF proof search, if we focus on a universally quantified assumption then we find C through unification. Focusing on the right, i.e. LF proof search, can only be applied if the goal is of box-type. Focusing on the left is standard and commences once we have decomposed the focused formula into its atoms as in the blur rule. At this point, we add the atomic formula to Γ and restart the process from the uniform left stage (to unbox the atomic formula if necessary). In practice, we implement backtracking when focusing. If for example, we cannot find a proof while focusing on the right, we backtrack and try focusing on the left. In practice we also support recursive types, which we treat as atomic computation-level types. These goal types may only be solved by focusing on the left.

We show in [31] that the focusing calculi in Figures 3 and 4 are sound and complete with respect to the sequent calculi presented in Section 2.2.2. The completeness result in particular is

$\boxed{\Delta;\Gamma \overset{R}{\Longrightarrow} \tau}$ There is a uniform right proof of τ in the focusing calculus

$$\frac{\Delta;\Gamma,y:\tau_1 \overset{\mathrm{R}}{\Longrightarrow} \tau_2}{\Delta;\Gamma \overset{\mathrm{R}}{\Longrightarrow} \tau_1 \to \tau_2} \to R \qquad \frac{\Delta,X:U;\Gamma \overset{\mathrm{R}}{\Longrightarrow} \tau}{\Delta;\Gamma \overset{\mathrm{R}}{\Longrightarrow} \Pi^{\Box} X{:}U.\tau} \Pi^{\Box} R \qquad \frac{\Delta;\cdot \gg \Gamma \overset{\mathrm{L}}{\Longrightarrow} \lceil \Psi \vdash P \rceil}{\Delta;\Gamma \overset{\mathrm{R}}{\Longrightarrow} \lceil \Psi \vdash P \rceil} \text{left to right}$$

$\boxed{\Delta;\Gamma \gg \Gamma' \overset{L}{\Longrightarrow} \lceil \Psi \vdash P \rceil}$ There is a uniform left proof of $\lceil \Psi \vdash P \rceil$ in the focusing calculus

$$\frac{\Delta,X:(\Phi \vdash Q);\Gamma \gg \Gamma' \overset{\mathrm{L}}{\Longrightarrow} \lceil \Psi \vdash P \rceil}{\Delta;\Gamma \gg \Gamma',y:\lceil \Phi \vdash Q \rceil \overset{\mathrm{L}}{\Longrightarrow} \lceil \Psi \vdash P \rceil} \Box L \qquad \frac{\tau \neq \lceil \Phi \vdash Q \rceil \quad \Delta;\Gamma,y:\tau \gg \Gamma' \overset{\mathrm{L}}{\Longrightarrow} \lceil \Psi \vdash P \rceil}{\Delta;\Gamma \gg \Gamma',y:\tau \overset{\mathrm{L}}{\Longrightarrow} \lceil \Psi \vdash P \rceil} \text{shift}$$

$$\frac{\Delta;\Psi \overset{u}{\Longrightarrow} P}{\Delta;\Gamma \gg \cdot \overset{L}{\Longrightarrow} \lceil \Psi \vdash P \rceil} \text{level} \qquad \frac{\Gamma(y) = \tau \quad \Delta;\Gamma > y:\tau \Rightarrow \lceil \Psi \vdash P \rceil}{\Delta;\Gamma \gg \cdot \overset{L}{\Longrightarrow} \lceil \Psi \vdash P \rceil} \text{focus to uniform}$$

$\boxed{\Delta;\Gamma > y:\tau \Rightarrow \lceil \Psi \vdash P \rceil}$ There is a focused proof of $\lceil \Psi \vdash P \rceil$ with focus on y in the focusing calculus

$$\frac{\Delta \Vdash C \Leftarrow U \quad \Delta;\Gamma > y':[\![C/X]\!]\tau \Rightarrow \lceil \Psi \vdash P \rceil}{\Delta;\Gamma > y:\Pi^{\Box} X{:}U.\tau \Rightarrow \lceil \Psi \vdash P \rceil} \Pi^{\Box} L$$

$$\frac{\Delta;\Gamma \overset{R}{\Longrightarrow} \tau_1 \quad \Delta;\Gamma > y':\tau_2 \Rightarrow \lceil \Psi \vdash P \rceil}{\Delta;\Gamma > y:\tau_1 \to \tau_2 \Rightarrow \lceil \Psi \vdash P \rceil} \to L \qquad \frac{\Delta;\cdot \gg \Gamma,y':\lceil \Phi \vdash Q \rceil \overset{L}{\Longrightarrow} \lceil \Psi \vdash P \rceil}{\Delta;\Gamma > y':\lceil \Phi \vdash Q \rceil \Rightarrow \lceil \Psi \vdash P \rceil} \text{blur}$$

Figure 4: Focusing calculus for the computation logic.

interesting as it requires an intermediate result stating that it does not matter in which context box-type assumptions appear. That is, if there is a proof in our sequent calculus of $\lceil \Psi \vdash P \rceil$ (possibly) using some assumption $y:\lceil \Phi \vdash Q \rceil$ in Γ then there is also a proof of $\lceil \Psi \vdash P \rceil$ where y is omitted but under the added assumption $X:(\Phi \vdash Q)$ in Δ. Given a Δ and Γ, $\Gamma^{-}_{\Delta,\Gamma}$ denotes Γ without any assumptions of box-type, and $\Delta^{+}_{\Delta,\Gamma}$ denotes Δ extended with the (unboxed) boxed assumptions from Γ.

Theorem 1 (Soundness)**.**
a) If $\Delta;\Psi \overset{u}{\Longrightarrow} A$ *then* $\Delta;\Psi \Longrightarrow A$

b) If $\Delta;\Psi \overset{u}{\Longrightarrow} \sigma:\Phi$ *then* $\Delta;\Psi \Longrightarrow \sigma:\Phi$

c) If $\Delta;\Psi > x:A \rightrightarrows P$ *then* $\Delta;\Psi,x:A \Longrightarrow P$
d) If $\Delta > X:U;\Psi \rightrightarrows P$ *then* $\Delta,X:U;\Psi \Longrightarrow P$
e) If $\Delta;\Gamma \overset{R}{\Longrightarrow} \tau$ *then* $\Delta;\Gamma \Longrightarrow \tau$

f) If $\Delta;\Gamma \gg \Gamma' \overset{L}{\Longrightarrow} \lceil \Psi \vdash P \rceil$ *then* $\Delta;\Gamma,\Gamma' \Longrightarrow \lceil \Psi \vdash P \rceil$
g) If $\Delta;\Gamma > y:\tau \Rightarrow \lceil \Psi \vdash P \rceil$ *then* $\Delta;\Gamma,y:\tau \Longrightarrow \lceil \Psi \vdash P \rceil$

Theorem 2 (Completeness)**.**
a) If $\Delta;\Psi \Longrightarrow A$ *then* $\Delta;\Psi \overset{u}{\Longrightarrow} A$

b) If $\Delta;\Psi \Longrightarrow \sigma:\Phi$ *then* $\Delta;\Psi \overset{u}{\Longrightarrow} \sigma:\Phi$

c) If $\Delta;\Psi \Longrightarrow P$ *then* $\Delta;\Psi > x : A \rightrightarrows P$ *for some* $A \in \Psi$ *or* $\Delta > X : U;\Psi \rightrightarrows P$ *for some* $U \in \Delta$
d) If $\Delta;\Gamma \Longrightarrow \tau$ *then* $\Delta^+_{\Delta,\Gamma};\Gamma^-_{\Delta,\Gamma} \stackrel{R}{\Longrightarrow} \tau$
e) If $\Delta;\Gamma \Longrightarrow \lceil \Psi \vdash P \rceil$ *then* $\Delta^+_{\Delta,\Gamma};\Gamma^-_{\Delta,\Gamma} \gg \cdot \stackrel{L}{\Longrightarrow} \lceil \Psi \vdash P \rceil$
f) If $\Delta;\Gamma \Longrightarrow \lceil \Psi \vdash P \rceil$ *then* $\Delta^+_{\Delta,\Gamma};\Psi \stackrel{u}{\Longrightarrow} P$ *or* $\Delta^+_{\Delta,\Gamma};\Gamma^-_{\Delta,\Gamma} > y : \tau \Rightarrow \lceil \Psi \vdash P \rceil$ *for some* $y : \tau \in \Gamma^-_{\Delta,\Gamma}$

The soundness proofs proceed by mutual structural induction on the given derivations and are straightforward. Proving completeness however is more involved and depends on several minor lemmas, consisting mostly of postponement results and a variation of the reflect rule for computation-level focusing, all proven with straightforward induction. The completeness proof proceeds again by mutual structural induction on the given derivations. In part a) the base case (i.e. the derivation is init$^\Psi$) is simplified by showing that the init$^\Psi$ rule is admissible under the addition of the rule $\Delta;\Psi, x : P \Longrightarrow P$. The remaining cases are then straightforward. Part b) is trivial. Part c) must be done by case analysis on the induction hypothesis but is otherwise straightforward. The base case in part d) is also made simpler by showing that the init$^\Gamma$ rule is admissible under the added assumption $\Delta;\Gamma, y : \lceil \Psi \vdash P \rceil \Longrightarrow \lceil \Psi \vdash P \rceil$. The cases in part d) where the derivation is either $\rightarrow R$, $\rightarrow L$, or $\Pi^\square L$ must be done by case analysis on whether or not τ_1, τ_2, or $[\![C/X]\!]\tau'$ (respectively) are atomic box-types. This is because the result of $\Delta^+_{\Delta,\Gamma}$ and $\Gamma^-_{\Delta,\Gamma}$ in the induction hypotheses depends on whether or not these assumptions are boxed. If they are boxed they will appear unboxed in $\Delta^+_{\Delta,\Gamma}$ and omitted from $\Gamma^-_{\Delta,\Gamma}$, otherwise they remain in $\Gamma^-_{\Delta,\Gamma}$. Part e) also involves case analysis on some subgoals, depending on if the assumptions in Γ are boxed or not. Some cases in part f) involve multiple case analyses- once on the result of the induction hypothesis, and once on the shape of the assumptions in Γ. The entire proof is given in [31].

3 Automation tactic

We present an overview of our automation tactic which implements the focusing calculi from Section 2.3. We begin with a walk-through of the theorem prover followed by a summary of its performance on various case studies from PL theory.

3.1 Example

To demonstrate the capabilities of the (meta-)theorem provers we examine how Beluga automatically proves that reduction is closed under expansion, a key lemma needed to show weak-head normalization for the simply-typed lambda-calculus. Specifically, we prove that if term `M` steps to term `N` and `N` reduces to type `A`, then `M` does as well.

Building upon the theory presented in Section 2.1, we encode the notion of reducibility using a logical relation by categorizing terms by the type they reduce to. We formalize the predicate in Beluga's reasoning logic, as it requires a strong, computational function space unlike the weak function space of LF. Inductive properties about contextual objects are defined using indexed recursive types [8]. We encode the set of reducible terms using the recursive type `Reduce`, which is stratified by its index `tp` [16].

```
stratified Reduce : {A:[ ⊢ tp]}{M:[ ⊢ term A]} ctype =
| I   : [ ⊢ halts M] → Reduce [ ⊢ b ] [ ⊢ M]
| Arr : [ ⊢ halts M]
          → ({N:[ ⊢ term A]} Reduce [ ⊢ A] [ ⊢ N] → Reduce [ ⊢ B ] [ ⊢ app M N])
            → Reduce [ ⊢ arr A B ] [ ⊢ M]
;
```

To begin our theorem, we load the Beluga file containing our LF and computation-level signatures into Harpoon. Harpoon then takes as input a theorem name and statement, and index of induction variable specified by its position in the overall goal, counted from left-to-right:

```
Name of theorem: bwd_closed
Statement of theorem: {A:[ ⊢ tp]} {M: [ ⊢ term A]} {M': [ ⊢ term A]} [ ⊢ step M M']
                        → Reduce [ ⊢ A] [ ⊢ M'] → Reduce [ ⊢ A] [ ⊢ M]
Induction order (empty for none): 1
```

Once loaded, users may simply invoke the tactic `auto`[1] to prove the lemma. Behind the scenes, Beluga gives the specified induction variable to our solver which then performs induction on said variable and generates the respective induction hypotheses for each subgoal. It then performs a round of inversions and bounded depth-first proof search on each produced subgoal. Users have the option of choosing their own depth bound by providing an argument to `auto`, otherwise it is set to 3. The proof search algorithm closely implements the focusing calculus presented in Section 2.3. As it searches for a proof, the solver also constructs proof terms in the form of LF and computation-level terms, and LF substitutions. If the solver can prove each case, the constructed proof term is presented to the user. If it cannot prove all subgoals, users will have to manually split on the specified variable (with the `split` tactic) and then call `auto` on each applicable case. In these instances, since no induction variable is specified, the solver will only perform inversions and bounded search. We show below the proof script that is being generated by invoking only the `auto` tactic.

```
1  proof bwd_closed : {A : ( ⊢ tp)} {M : ( ⊢ term A)}{M' : ( ⊢ term A)}
2                     [ ⊢ step M M'] → Reduce [ ⊢ A] [ ⊢ M'] → Reduce [ ⊢ A] [ ⊢ M] =
3  / total 1 /
4  intros
5  { A : ( ⊢ tp), M : ( ⊢ term A), M' : ( ⊢ term A)
6  | x : [ ⊢ step M M'], x1 : Reduce [ ⊢ A] [ ⊢ M']
7  ; solve
8      let [ ⊢ Y] = x in
9      case [ ⊢ A] of
10     | [ ⊢ b] =>
11       let (I x2 : Reduce [ ⊢ b] [ ⊢ M']) = x1 in
12       let ([ ⊢ halts/m X3 X4] : [ ⊢ halts M']) = x2 in
13       let [ ⊢ val/c] = [ ⊢ X4] in I [ ⊢ halts/m (sstep Y X3) val/c]
14     | [ ⊢ arr X X1] =>
15       let (Arr x2 x3 : Reduce [ ⊢ arr X X1] [ ⊢ M']) = x1 in
16       let ([ ⊢ halts/m X5 X6] : [ ⊢ halts M']) = x2 in
17       let [ ⊢ val/abs ] = [ ⊢ X6] in
18       Arr [ ⊢ halts/m (sstep Y X5) (val/abs )]
19         (mlam N => fn y =>
20           bwd_closed [ ⊢ X1] [ ⊢ app M N] [ ⊢ app M' N] [ ⊢ stepapp Y] (x3 [ ⊢ N] y)) } ;
```

[1] In Harpoon, this tactic is invoked using `inductive-auto-solve`. See https://beluga-lang.readthedocs.io/en/latest/harpoon/proof-automation.html#inductive-auto-solve for a description on how to use the tactic.

The BELUGA program that our solvers construct is given in lines 8 - 20, which gets spliced into the overall proof script. From the given proof script, we can extract a complete BELUGA program [11]. Previously, this lemma could be proved with no less than 10 HARPOON tactic calls (see also [11]).

The proof begins with the automatic application of the `intros` tactic, which performs the uniform right phase. The rest of the proof is constructed using BELUGA's theorem provers. It first unboxes `x` (line 8), concluding the uniform left phase. Since an induction variable was specified, the algorithm immediately splits on the variable. After a split, there is a round of inversions for each produced subgoal followed by bounded focused proof search. In the first case (`A = b`) after the inversions (line 13), the computation-level solver focuses on the constructor `I` and solves its subgoal by providing the appropriate boxed LF proof term (`halts/m (sstep Y X3) val/c`) which it finds again through focused proof search, but this time on the LF level. In the second case (`A = arr X X1`), the solver focuses on `Arr` which requires solving two subgoals, one of which is of function type. It is here where the solver focuses on the induction hypothesis to solve a subgoal. If the solver focuses on the "wrong" assumption, it will execute backtracking and continue focusing until it has exhausted all possible branches on the bounded search tree.

3.2 Evaluation

The proof-search procedures behind our automation tactic is the beginning of the implementation of the focusing calculi presented in Section 2.3. There are several areas of incompleteness that may be improved in the future [31]. Nevertheless, the tactic is able to prove a number of interesting theorems both semi- and fully-automatically. We provide a summary of these case studies here.

Case study	Automation	Difficulty	Interesting proof features
MiniML/fix type preservation	Full	Advanced	Solving substitutions, I.H. appeal, inversions
MiniML/fix value soundness	Full	Basic	I.H. appeal
STLC weak-head normalization lemmas	Full	Intermediate	Inversions, I.H. appeal, higher-order solving
STLC type uniqueness	Partial	Basic	I.H. appeal, inversions
Untyped λ-calculus reduction lemmas	Full/Partial	Basic	I.H. appeal

Table 1: Overview of case studies.

We categorize our case studies by the amount of automation that may be successfully applied. We say full automation is applied when only `auto` is used to complete a proof (as in the previous example). Partial automation is used on induction proofs when not all the subgoals fall within the prover's applicable subset. In these instances, `auto` is used after a variable split has been manually made, and only on a portion of the subgoals. All proofs proceed by induction along with various features that we have outlined.

We use our tactic to prove key lemmas required to show weak-head normalization for the simply-typed lambda-calculus. The backwards closed lemma is particularly interesting as it requires higher-order function type solving. Type uniqueness for the simply-typed lambda-calculus

can be proven semi-automatically as two of its cases involve parameter variables and context block schemas. We prove all lemmas regarding ordinary reduction for the untyped lambda-calculus automatically, and all but one lemma regarding parallel reduction automatically. These lemmas are used to prove equivalence of the ordinary and parallel reductions, and ultimately the Church-Rosser theorem for each reduction procedure. For MiniML without fixpoints, we are able to prove type preservation and value soundness fully-automatically. Preservation in particular showcases the solvers ability to solve for substitutions [31].

Beluga's proving power does not yet surpass that of Twelf's. However, Beluga is able to reason directly using logical relations, unlike Twelf. Certain properties, like normalization theorems, are most commonly proven using logical relations. In Twelf, users must find alternative proof methods [30, 1], which may be conceptually different from on-paper formulations and require more work from the user to construct additional machinery. In Beluga, such logical relations may be directly translated from on-paper formulations and their proofs become simplified with the use of our automation tactic.

4 Related work

4.1 Proof environments based on HOAS

Twelf [22] currently provides the most automation out of all HOAS-based proof assistants designed to formalize PLs. It fully automatically proves many theorems such as type-preservation for MiniML, Church-Rosser for the simply typed lambda-calculus, and cut-admissibility for first-order logic [29, 28]. It essentially has a simple loop that splits on an assumption based on a heuristic, generates induction hypothesis, and tries to prove a given goal by bounded depth-first proof search. If the last step fails, it will again split on an assumption heuristically picking one. This has proven remarkably effective. However, the kind of properties that can be stated is more restrictive. For example, Twelf's meta-logic permits only theorems expressed as Π_2 statements and lacks support for recursive types. Therefore it cannot, for example, directly deal with proofs that proceed via logical relations. More importantly, Twelf's prover does not produce proof terms and therefore provides no way to verify its proofs. Even worse, if the meta-theorem prover fails to find a proof automatically, there is no possibility for the user to pick up the pieces and manually proceed.

Abella is another HOAS-based proof assistant with the capabilities to mechanize meta-theoretic proofs about PLs [14]. Semi-automation exists in Abella provided in the form of tactics that are similar to the ones previously implemented in Harpoon. Contexts and (simultaneous) substitutions in Abella are not treated as first-class like they are in Beluga. This means that in order to use such constructs, users must manually define them and any properties they wish to utilize about them (e.g. context weakening). These can further add to the complexity of formal theorem proving. As Twelf, Abella also does not construct proof objects during proof development. Moreover, the tactics used in constructing proofs in Abella concentrate on "small" steps and there is presently no analogue to the proof search tactic that we describe in this paper. However, we believe that similar semi-automation could be added to Abella to discharge straightforward cases.

4.2 General proof environments

The Hybrid system aims to bring full HOAS support to general proof environments like Coq and Isabelle/HOL [2, 18]. These systems often have more users compared to specialized systems and thus more reason to provide tooling and automation. Both systems employ the use of tacticals, which allow users to create their own tactics specific to their mechanizations.

In Coq, users write decision procedures in a tactic language like LTac [10]. There are general automated proof search tactics (auto, eauto, iauto and jauto) but they do not conduct any case analysis (including inversions), inductions, or rewritings, and are intended to finish a proof instead of complete it entirely [26]. Coq also offers many libraries to assist with building and verifying infrastructure needed in PL mechanizations. Autosubst may be used as a library for example to automatically generate parallel substitution operations for a custom type and prove the related substitution lemmas [27]. The libraries Ott [32] and LNgen [4] may be used together to generate locally nameless definitions from a specification and provide the corresponding recursion schemes and infrastructure lemmas. However these tools are limited as they only automate trivial lemmas.

For much of its automation, Isabelle elicits the help of several external solvers. Sledgehammer [21], for example, takes a goal and heuristically chooses from Isabelle's libraries containing various lemmas, definitions, and axioms, a few hundred applicable ones to perform search over. Then, translates the goal and each of these assumptions to SMT (first-order logic) and sends the query off to an external SMT or resolution-based solver. In its own system, Isabelle performs various general-purpose proof search methods which help discharge simple parts of a proof allowing users to focus on the main ones [7]. They also have several strengthened endgame tactics which are meant to finish a proof but provide no hints upon failing.

Despite the abundance of tooling in these systems, some specialized systems (Beluga and Twelf for example) offer more automation for PL mechanizations. This is because these systems have fixed specification logics, so automatic proof procedures can be more intricate. Hybrid on the other hand allows for encodings of various specification logics, therefore proof procedures are more difficult as they must be customized to work for different logics.

5 Conclusion

In closing, we have presented the theorem and meta-theorem provers behind Beluga, a specialized proof assistant with sophisticated built-in support for specifying formal systems. These provers perform two-leveled proof search over a core subset of Beluga's logic which allows for the automatic completion of many simple lemmas and cases of PL theory proofs. Users of Beluga may now bypass simple proofs and focus their energy on the interesting cases. Along with our implementation, we provide a theoretical foundation for our solvers in the form of a cut-free sequent calculus, which is easy to understand, and a sound and complete focusing calculus, which closely reflects our implementation. These provide us with a way to study our implementation and ensure its correctness.

Our next steps are to expand the solver so that its proving capabilities are equivalent to that of the logic presented in this paper [31]. After that we plan to add support for context block schemas, and substitution and parameter variables, which should bring its proving power up to that of Twelf's. Finally, it would be interesting to extend the focusing calculus (and implementation) with automatic induction, similar to [5].

References

[1] Andreas Abel (2008): *Normalization for the Simply-Typed Lambda-Calculus in Twelf.* In: *LFM '04*, 199, pp. 3–16, doi:10.1016/j.entcs.2007.11.009.

[2] Simon Ambler, Roy L. Crole & Alberto Momigliano (2002): *Combining Higher Order Abstract Syntax with Tactical Theorem Proving and (Co)Induction.* In: *Proc. TPHOLs '02*, *LNCS* 2410, Springer-Verlag, p. 13–30, doi:10.1007/3-540-45685-6_3.

[3] Jean-Marc Andreoli (1992): *Logic Programming with Focusing Proofs in Linear Logic.* *J. Log. Comput.* 2(3), pp. 297–347, doi:10.1093/logcom/2.3.297.

[4] Brian E. Aydemir & Stephanie Weirich (2010): *LNgen: Tool Support for Locally Nameless Representations.* Technical Report, University of Pennsylvania. Available at `https://repository.upenn.edu/cis_reports/933`.

[5] David Baelde, Zachary Snow & Dale Miller (2010): *Focused Inductive Theorem Proving.* In Jürgen Giesl & Reiner Haehnle, editors: *IJCAR'10*, LNAI 6173, Springer, pp. 278–292, doi:10.1007/978-3-642-14203-1_24.

[6] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series (TTCS), Springer Berlin, Heidelberg, doi:10.1007/978-3-662-07964-5.

[7] Jasmin Christian Blanchette, Lukas Bulwahn & Tobias Nipkow (2011): *Automatic Proof and Disproof in Isabelle/HOL.* In Cesare Tinelli & Viorica Sofronie-Stokkermans, editors: *FroCoS*, *LNAI* 6989, Springer Berlin Heidelberg, pp. 12–27, doi:10.1007/978-3-642-24364-6_2.

[8] Andrew Cave & Brigitte Pientka (2012): *Programming with Binders and Indexed Data-Types.* In: *Proc. POPL '12*, ACM, p. 413–424, doi:10.1145/2103656.2103705.

[9] Andrew Cave & Brigitte Pientka (2018): *Mechanizing proofs with logical relations – Kripke-style.* *Math. Struct. Comput. Sci.* 28(9), pp. 1606 – 1638, doi:10.1017/S0960129518000154.

[10] David Delahaye (2000): *A Tactic Language for the System Coq.* In Michel Parigot & Andrei Voronkov, editors: *LPAR*, Springer Berlin Heidelberg, pp. 85–95, doi:10.1007/3-540-44404-1_7.

[11] Jacob Errington, Junyoung Jang & Brigitte Pientka (2021): *Harpoon: Mechanizing Metatheory Interactively: (System Description).* In André Platzer & Geoff Sutcliffe, editors: *Automated Deduction - CADE 28*, *LNAI* 12699, Springer, Cham, pp. 636–648, doi:10.1007/978-3-030-79876-5_38.

[12] Amy Felty, Alberto Momigliano & Brigitte Pientka (2015): *The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations.* *J. Autom. Reasoning* 55(4), p. 307–372, doi:10.1007/s10817-015-9327-3.

[13] Amy Felty & Brigitte Pientka (2010): *Reasoning with Higher-Order Abstract Syntax and Contexts: A Comparison.* In: *Proc. ITP '10*, *LNCS* 6172, Springer-Verlag, p. 227–242, doi:10.1007/978-3-642-14052-5_17.

[14] Andrew Gacek (2008): *The Abella Interactive Theorem Prover (System Description).* In: *IJCAR'08*, *LNAI* 5195, Springer Berlin Heidelberg, pp. 154–161, doi:10.1007/978-3-540-71070-7_13.

[15] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics.* *Journal of the ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[16] Rohan Jacob-Rao, Brigitte Pientka & David Thibodeau (2018): *Index-Stratified Types.* In Hélène Kirchner, editor: *FSCD 2018*, *LIPIcs* 108, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 19:1–19:17, doi:10.4230/LIPIcs.FSCD.2018.19.

[17] Dale A. Miller, Gopalan Nadathur, Frank Pfenning & Andre Scedrov (1991): *Uniform Proofs as a Foundation for Logic Programming.* *Ann. Pure Appl. Log.* 51, pp. 125–157, doi:10.1016/0168-0072(91)90068-W.

[18] Alberto Momigliano, Alan J. Martin & Amy P. Felty (2008): *Two-Level Hybrid: A System for Reasoning Using Higher-Order Abstract Syntax.* In: *Proc. LFMTP '07*, *ENTCS* 196, pp. 85–93, doi:10.1016/j.entcs.2007.09.019.

[19] Aleksandar Nanevski, Frank Pfenning & Brigitte Pientka (2008): *Contextual Modal Type Theory. TOCL* 9(3), doi:10.1145/1352582.1352591.

[20] Lawrence C. Paulson (1994): *Isabelle: A Generic Theorem Prover.* Springer-Verlag, doi:10.1007/BFb0030541.

[21] Lawrence C. Paulson & Kong Woei Susanto (2007): *Source-Level Proof Reconstruction for Interactive Theorem Proving.* In Klaus Schneider & Jens Brandt, editors: *TPHOLs '07*, *LNCS* 4732, Springer Berlin Heidelberg, pp. 232–245, doi:10.1007/978-3-540-74591-4_18.

[22] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf — A Meta-Logical Framework for Deductive Systems.* In H. Ganzinger, editor: *CADE-16*, *LNAI* 1632, Springer, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[23] Brigitte Pientka (2008): *A Type-Theoretic Foundation for Programming with Higher-Order Abstract Syntax and First-Class Substitutions.* In George C. Necula & Philip Wadler, editors: *Proc. POPL '08*, 43, ACM, p. 371–382, doi:10.1145/1328897.1328483.

[24] Brigitte Pientka & Jana Dunfield (2008): *Programming with proofs and explicit contexts.* In: *Proc. PPDP '08*, PPDP '08, ACM, pp. 163–173, doi:10.1145/1389449.1389469.

[25] Brigitte Pientka & Jana Dunfield (2010): *Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description).* In Jürgen Giesl & Reiner Hähnle, editors: *IJCAR '10*, *LNAI* 6173, Springer Berlin, Heildelberg, pp. 15–21, doi:10.1007/978-3-642-14203-1_2.

[26] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach & Brent Yorgey (2022): *Programming Language Foundations.* *Software Foundations* 2, Electronic textbook. Available at `http://softwarefoundations.cis.upenn.edu`. Version 6.2.

[27] Steven Schäfer, Tobias Tebbi & Gert Smolka (2015): *Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions.* In Christian Urban & Xingyuan Zhang, editors: *ITP*, Springer International Publishing, pp. 359–374, doi:10.1007/978-3-319-22102-1_24.

[28] Carsten Schürmann (2000): *Automating the Meta Theory of Deductive Systems.* Ph.D. thesis, Carnegie Mellon University. Available at `https://www.cs.cmu.edu/~rwh/students/schuermann.pdf`.

[29] Carsten Schürmann & Frank Pfenning (1998): *Automated Theorem Proving in a Simple Meta-Logic for LF.* In Claude Kirchner & Hélène Kirchner, editors: *Proc. CADE-15*, Springer-Verlag LNCS, pp. 286–300, doi:10.1007/BFb0054266.

[30] Carsten Schürmann & Jeffrey Sarnat (2008): *Structural Logical Relations.* In: *LICS*, IEEE Computer Society, pp. 69–80, doi:10.1109/LICS.2008.44.

[31] Johanna Schwartzentruber (2023): *Semi-Automation of Meta-Theoretic Proofs.* Master's thesis, McGill University.

[32] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar & Rok Strniša (2007): *Ott: Effective Tool Support for the Working Semanticist.* In: *Proc. ICFP '07*, ICFP '07, ACM, p. 1–12, doi:10.1145/1291151.1291155.

[33] Kevin Watkins, Iliano Cervesato, Frank Pfenning & David Walker (2004): *A Concurrent Logical Framework: The Propositional Fragment.* In Stefano Berardi, Mario Coppo & Ferruccio Damiani, editors: *TYPES*, Springer Berlin Heidelberg, pp. 355–377, doi:10.1007/978-3-540-24849-1_23.

Parallel Verification of Natural Deduction Proof Graphs

James T. Oswald Brandon Rozek

Rensselaer *AI* & Reasoning (RAIR) Lab,
Rensselaer Polytechnic Institute (RPI)
Troy, New York, USA

{oswalj, rozekb}@rpi.edu

Graph-based interactive theorem provers offer a visual representation of proofs, explicitly representing the dependencies and inferences between each of the proof steps in a graph or hypergraph format. The number and complexity of these dependency links can determine how long it takes to verify the validity of the entire proof. Towards this end, we present a set of parallel algorithms for the formal verification of graph-based natural deduction ($\mathscr{ND}$) style proofs. We introduce a definition of layering that captures dependencies between the proof steps (nodes). Nodes in each layer can then be verified in parallel as long as prior layers have been verified. To evaluate the performance of our algorithms on proof graphs, we propose a framework for finding the performance bounds and patterns using directed acyclic network topologies (DANTs). This framework allows us to create concrete instances of DANTs for empirical evaluation of our algorithms. With this, we compare our set of parallel algorithms against a serial implementation with two experiments: one scaling the problem size and the other scaling the number of threads. Our findings show that parallelization results in improved verification performance for certain DANT instances. We also show that our algorithms scale for certain DANT instances with respect to the number of threads.

1 Introduction

A major role of an interactive theorem prover is to take an existing proof and verify that it is valid with respect to the logical calculi used. This involves iterating over each step of a proof and verifying both that it syntactically matches the transformation of the formula under the rule, and that it is valid with respect to the semantics of the underlying proof system. Interactive theorem provers such as Coq [19], Lean [14], and HyperSlate [4] verify not only that the proof written by the user is correct, but also every underlying proof that the given proof depends on. This generally amounts to verifying large portions of the standard library and other popular libraries such as mathlib [6]. Our work makes a step toward speeding up the proof verification process. We focus on the verification of natural deduction proof graphs, such as those represented in HyperSlate, though the ideas from this approach could be adapted to other interactive theorem provers as well.

In order to speed up verification, we look toward parallel computing. One naive implementation would be to verify all the proof steps in parallel. This assumes, however, that the step has all the semantic information needed to show validity. This is often not the case for many logic calculi. Assumptions are introduced and discharged in the case of natural deduction. Variables may be assigned to constants. These issues present a constraint that in order to parallelize verification, we need to ensure that some steps are verified before others. We achieve this by introducing a layering approach. Given a definition of layering that induces a topological partial order, every step within a layer n only depends on steps within the layers prior. Given these layers, we can then verify all the steps that are from the same layer in parallel without worrying about invalidating the underlying semantics. To illustrate this approach, we present the parallel verification of natural deduction proof graphs.

A. Ciaffaglione and C. Olarte (Eds.): Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2023)
EPTCS 396, 2023, pp. 36–51, doi:10.4204/EPTCS.396.4

© J. Oswald and B. Rozek
This work is licensed under the
Creative Commons Attribution License.

The underlying dependency nature of each of the steps induces a directed acyclic hypergraphical representation where nodes hold $\mathscr{ND}$ statements and hyperedges between nodes represent inference rules. This graphical representation not only gives us an easy way to visualize such proofs, but also provides insight on empirically validating our parallel algorithms. Inspired by computer network topologies, we introduce directed acyclic network topologies (DANTs) as a way to identify classes of graphical proofs. These topologies provide a method of comparing the performance of different verification strategies on various proof structures.

The contributions of this work are as follows: (1) A layering approach that decouples the dependencies of proof steps within $\mathscr{ND}$ proofs. (2) Parallel verification algorithms that outperform serial verification on non-straight topologies and scales with the number of hardware-based threads. (3) Introduction of several classes of graphical proofs, with an eye on empirical evaluation. The relevant background which includes $\mathscr{ND}$ and hypergraphical representations is discussed in §2. Within §3, we discuss the proof verification algorithm and several optimizations. Then in §4, we discuss directed acyclic network topologies to empirically evaluate common proof structures. In that section, we also discuss our performance and scaling results. We then conclude by talking about related work in §5.

2 Background

2.1 Natural Deduction

Natural deduction ($\mathscr{ND}$) is a logic calculus independently proposed in [10, 12] in an effort to emulate human-level reasoning through assumptions and chains of inference. There are many different styles of proof that fall under natural deduction, the three most common come from Gentzen [10], Jaśkowski [12], and Fitch [9]. However, we are mainly interested in a style that interoperates with a hypergraphical representation of natural deduction proofs.

$$\frac{}{\{\phi\} \vdash \phi}\ \mathrm{A} \qquad \frac{\Gamma \vdash \psi \quad \Sigma \vdash \phi}{\Gamma \cup \Sigma \vdash \phi \wedge \psi}\ \wedge I \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi}\ \wedge E_l \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}\ \wedge E_r$$

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \psi \vee \phi}\ \vee I_l \qquad \frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi}\ \vee I_r \qquad \frac{\Delta \vdash \psi \vee \phi \quad \Gamma \cup \{\psi\} \vdash \chi \quad \Sigma \cup \{\phi\} \vdash \chi}{\Delta \cup \Gamma \cup \Sigma \vdash \chi}\ \vee E$$

$$\frac{\Gamma \cup \{\phi\} \vdash \psi}{\Gamma \vdash \phi \rightarrow \psi}\ \rightarrow I \qquad \frac{\Gamma \vdash \phi \quad \Sigma \vdash \phi \rightarrow \psi}{\Gamma \cup \Sigma \vdash \psi}\ \rightarrow E$$

$$\frac{\Gamma \cup \{\phi\} \vdash \psi \quad \Sigma \vdash \neg\psi}{\Gamma \cup \Sigma \vdash \neg\phi}\ \neg I \qquad \frac{\Gamma \cup \{\neg\phi\} \vdash \psi \quad \Sigma \vdash \neg\psi}{\Gamma \cup \Sigma \vdash \phi}\ \neg E$$

$$\frac{\Gamma \cup \{\phi\} \vdash \psi \quad \Sigma \cup \{\psi\} \vdash \phi}{\Gamma \cup \Sigma \vdash \phi \leftrightarrow \psi}\ \leftrightarrow I \qquad \frac{\Gamma \vdash \phi \quad \Sigma \vdash \phi \leftrightarrow \psi}{\Gamma \cup \Sigma \vdash \psi}\ \leftrightarrow E_l \qquad \frac{\Gamma \vdash \psi \quad \Sigma \vdash \phi \leftrightarrow \psi}{\Gamma \cup \Sigma \vdash \phi}\ \leftrightarrow E_r$$

Figure 1: Our inference schemata for natural deduction. Within each schema, Γ, Σ, Δ are sets of formulae, and ϕ, ψ, χ are meta-logical variables which range over formulae. Note that our formulation of $\neg I, \neg E, \leftrightarrow I, \leftrightarrow E$ differs from those typically seen in other works such as [17] but are equivalent.

In this paper, we focus on propositional natural deduction. Let p denote an atomic proposition. The language of propositional logic may be defined inductively using Backus Naur Form (BNF) as the following:

$$\phi ::= p|\neg\phi|(\phi \land \phi)|(\phi \lor \phi)|(\phi \rightarrow \phi)|(\phi \leftrightarrow \phi)$$

Our inference rules for $\mathscr{ND}$ are summarized in Figure 1 [1]. This formalization is modeled after Bringsjord [3] and fully captures the notion of discharging of assumptions. It is also particularly well suited to hypergraphical representation, which will be discussed in §2.2. These inference rules can be broadly split into two categories: (1) introduction rules ($\land I, \lor I_l, \lor I_r, \neg I, \rightarrow I, \leftrightarrow I$), in which a logical connective is introduced into the conclusion, and (2) elimination rules ($\land E_l, \land E_r, \lor E, \neg E, \rightarrow E, \leftrightarrow E_l, \leftrightarrow E_r$), in which a connective in a rule's premise is removed in its conclusion. The outlier here is the Assumption rule (A) which allows us to assert $\{\phi\} \vdash \phi$, or in English, "assuming ϕ, ϕ follows".

For a natural deduction proof, a step is considered valid if the formula is well-formed and it is justified by a rule of inference. Valid formulae with no assumptions are called tautologies. A proof is considered valid iff all of its steps are valid. An example of a valid proof can be seen in Figure 2.

$$\dfrac{\dfrac{}{\{A \lor B\} \vdash A \lor B}\,\text{A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\{\neg A\} \vdash \neg A}\,\text{A} \quad \dfrac{}{\{\neg B\} \vdash \neg B}\,\text{A}}{\{\neg B, \neg A\} \vdash \neg A \land \neg B}\,\land I}{\{\neg B, \neg A\} \vdash \neg A}\,\land E_l \qquad \dfrac{}{\{A\} \vdash A}\,\text{A}}{\{\neg A, A\} \vdash B}\,\neg E \qquad \dfrac{}{\{B\} \vdash B}\,\text{A}}{\{A \lor B, \neg A\} \vdash B}\,\lor E$$

Figure 2: An example of a valid proof of B from $\{A \lor B, \neg A\}$. All steps are valid since at each step (1) all formulae are well formed and (2) the provided rule of inference can be legally applied at each stage given the current assumptions and premises.

2.2 Hypergraphical Representation

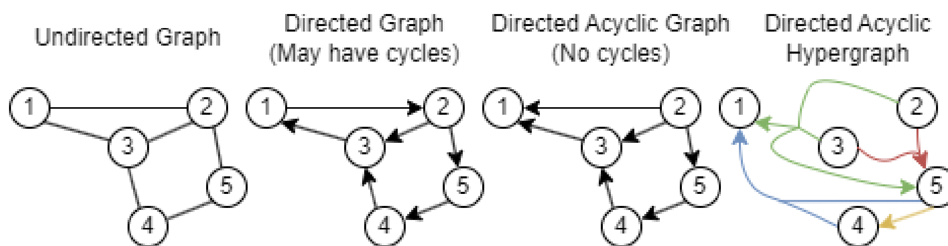


Figure 3: Visualizations of 4 different types of graphs, note that in the hypergraph, edges that share the same color are the same edge.

A natural deduction proof can be represented diagrammatically as a directed acyclic hypergraph [20, 1]. A directed acyclic hypergraph is a generalization of a directed acyclic graph (DAG) which is a

[1] While on the surface this formalization may appear similar to sequent natural deduction[16], we use "$\vdash$" in this formalism to mean syntactic entailment, with $\Gamma \vdash \phi$ being read as "Assuming Γ, then ϕ" or "ϕ can be derived from Γ".

mathematical structure (V,E) where V is a set of vertices and $E : (V \times V)$ is a set of pairs of vertices. Acyclic in this context means that for any vertex v, it is not possible to find a path following the directed edges that leads to v. Directed acyclic hypergraphs extend this by allowing a set of vertices to be connected to a set of vertices by a single edge, thus a directed acyclic hypergraph is a structure (V,E) where V is a set of vertices and $E : \mathscr{P}(V) \times \mathscr{P}(V)$, where $\mathscr{P}(V)$ is the power-set of the set of vertices. Figure 3 shows visualizations of the three graph formalisms described.

To represent natural deduction proofs as hypergraphs, vertices represent premises and conclusions, and edges represent inference rules. A *proof graph* will be defined as a hypergraph of the form (V,E) where V is a set of statements in the form $\Gamma \vdash \phi$ and $E : \mathscr{P}(V) \times V$ is the set of directed hypergraphical edges representing inference rules applied between statements in the proof.[2] This formalism underlies the representation of proofs in graphical interactive theorem provers such as [4, 15]. Figure 4 provides examples of two natural deduction proofs that have been converted to hypergraphical form.

Interactive theorem provers often do not force the user to keep track of proof state. Therefore, it is important to note that we are interested in verifying proof graphs where the assumptions on each node are yet to be known. We are only given the ϕ on each node and must compute the Γ based on how the assumptions update within the inference rules. If we had both Γ and ϕ, parallelization would be trivial, since we can then verify all the nodes in parallel.

Proof graphs implicitly provide additional useful features for representing collections of natural deduction proofs, particularly those that are commonly added onto natural deduction via additional formalisms. First, proof graphs provide the ability to compactly represent proofs that contain reoccurring subproofs. This is because each hyper-node may have multiple outgoing hyper-edges, representing multiple inferences it is used in. While this feature is implicitly captured by proof graphs, natural deduction proofs require an additional formalism allowing named theorems that can be used in other proofs to provide this functionally. Another feature is that a single proof graph can have multiple conclusions or even contain multiple proofs where each proof is a disjoint hyper-subgraph. Without proof graphs, the ability to represent this feature would require a formalism in which a set of proofs can be be treated as a single proof.

2.3 Multiprocessing

In this work we use a shared memory model for multiprocessing. This involves multiple threads independently operating over the same shared memory space. More specifically, we make heavy use of single program multiple data (SPMD) style programs. With a fixed number of threads instantiated, we attempt to distribute work evenly across all the threads. Our work makes use of two important concepts from multi-processing: thread-safety and reductions (see [13] for more extensive coverage). *Thread-safety* within a shared memory model is the notion that parallel algorithms are safe from errors due to concurrent writes and reads from the same piece of memory, known as a *race condition*. The second concept is the notion of a parallel reduction. A *parallel reduction* is an operator that takes a list of elements and computes a single element in parallel. A small example of this would be parallel sum over the list $(1,2,3,4)$: if we add 1 and 2 on one thread and 3 and 4 on another thread in parallel, and then sum their results, we can sum the entire list in 2 steps rather than the 3 steps it would take to sum the list in serial.

[2] Note we use $\mathscr{P}(V) \times V$ rather than $\mathscr{P}(V) \times \mathscr{P}(V)$ since for all inference rules enumerated in Figure 1 there is only one conclusion, thus each hyper-edge representing an inference rule will only ever have one outgoing connection.

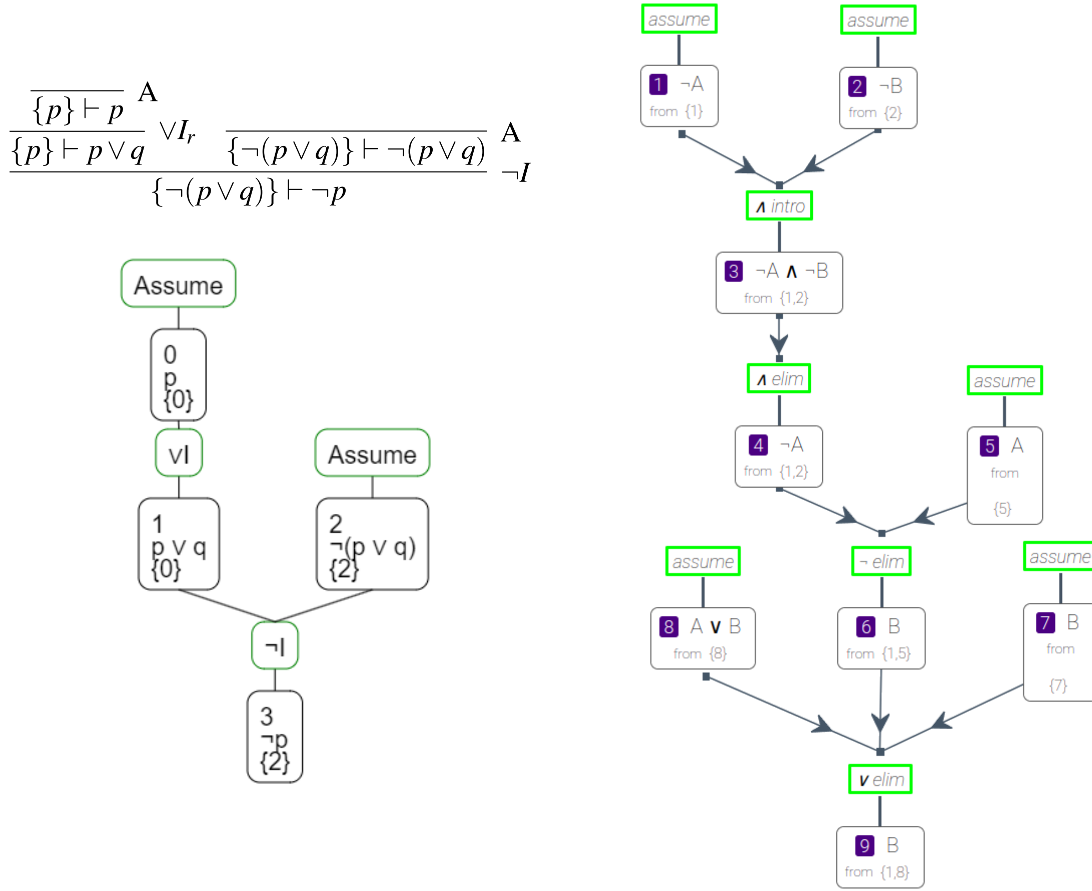


Figure 4: (Top Left) A valid natural deduction proof that $\{\neg(p \vee q)\} \vdash \neg p$ (Bottom Left) The same proof represented as a hypergraphical proof graph structure in the Lazyslate interactive theorem prover[15]. (Right) Proof graph of the proof from Figure 2 of $\{A \vee B, \neg A\} \vdash B$ in the HyperSlate [4] interactive theorem prover.

3 Approach

We mentioned in §2 that natural deduction makes use of assumptions and chains of inference in its proofs. In the hypergraphical representation, to show that a given node is valid, we need to show both that the *syntactic transformation* is valid and that the *assumption constraints* are met with respect to the justification used inside that proof. Let us consider the rule disjunction elimination (more commonly known as proof by cases) from Figure 1 and its usage in the right proof of Figure 4. For example, we wish to show that bottom node B is valid. For the syntactic check, we need to ensure that there are three parent nodes, one of them is a disjunct, and two of the other parent nodes match the current node. Then for the assumption constraints, we need to make sure that for one parent node B it has A in its assumption set, and for the other parent node B it has B in its assumption set. Note from our discussion of the hypergraphical representation in §2.2 that the underlying nodes do not contain the assumption information themselves, but they are computed by the application of each inference rule. This creates the need of an additional data structure that we call `assumptions` during the verification process. We obtain the justification of a given step by calling `just` on the node. This will return the justification that

is stored on the incoming edge of the node.

For a baseline comparison with the parallel verification algorithms, we designed a single-threaded implementation that shares the same algorithmic structure as the parallel ones minus the usage of shared memory and threading. The benchmark results are further discussed in §4. Our algorithm works by maintaining a global map of nodes to their set of assumptions. To ensure that a node does not get verified before its parent, we make use of a layering approach which induces a *topological partial ordering* on the nodes. A topological partial ordering, also referred to as *topological generations*, of a proof graph $G = (V, E)$ is a partial ordering $\preccurlyeq$ on the nodes V where for each hyperedge $(\{v_{i0}, \cdots, v_{in}\}, v_o)$, all incoming nodes $\{v_{i0}, \cdots, v_{in}\}$ appear before the outgoing node v_o, that is $\forall (V_i, v_o) \in E : \forall v_{ij} \in V_i : v_{ij} \preccurlyeq v_o$. This layering approach for generating a topological partial ordering is similar to the well known serial topological sort algorithm [7] which generates a topological linear ordering of the nodes but lacks parallelizability. Figure 5 provides a colored example of the nodes on each layer. More formally, we define node n to be on a layer $L(n)$ inductively as follows:

$$L(n) = \begin{cases} 0 & \text{if } n \text{ is an assumption} \\ 1 + \max_{m \in P(n)}(L(m)), & \text{otherwise} \end{cases} \quad (1)$$

where $P(n)$ maps a node to its parents.

3.1 Single-Threaded Implementation

Algorithm 1 Single-Threaded Algorithm

```
 1: procedure VERIFY(ProofGraph p)
 2:     Initialize assumptions to be empty.
 3:     Create set of nodes on each layer using Equation 1 and store in layerMap.
 4:     for layerNodes in layerMap do
 5:         for n in layerNodes do
 6:             justification = just(n)
 7:             ruleInfo = (m, assumptions(m)) ∀ m ∈ parents(n)
 8:             if not is_valid(n, justification, ruleInfo) then
 9:                 return false
10:             Update assumptions(n) using the justification and ruleInfo.
11:     return true
```

The full single-threaded procedure is described in Algorithm 1. For every layer, the procedure performs the following actions: (1) Verify that the node is valid with respect to the justification claimed using the node's and its parents' syntactic information and the parents' assumption information. (2) If valid, update the `assumptions` data structure for the current node based on the parents' assumptions and justification.

In order to better highlight the progression of the algorithm, we will walk through an example by looking at the verification of Figure 4 (Right). Subscripts for the propositions help to distinguish between formulae by referencing the ID denoted inside the purple box in the figure. In the beginning of the algorithm, the first layer only contains assumptions:

$$\texttt{currentLayer} = \{(\neg A)_1, (\neg B)_2, A_5, B_7, (A \vee B)_8\}$$

We then go through each node and verify them. Since they are justified as assumptions, they are trivially valid. The nodes then have their assumptions updated. The next layer only contains $(\neg A \wedge \neg B)_3$. This validates and the node's assumptions are updated to $\{(\neg A)_1, (\neg B)_2\}$. The third layer only contains $(\neg A)_4$. This validates and the assumptions are propagated forward. On the fourth layer, the node B_6 is justified by negation elimination. This validates and the node's assumptions are set to $\{(\neg A)_1, A_5\}$. The fifth and final layer only contains the node B_9. As the node is justified by disjunctive elimination and is valid, we update the assumptions to $\{(\neg A)_1, (A \vee B)_8\}$. As we have gone through all the layers successfully, the whole hypergraph is valid.

Theorem 1. *For all Proof Graphs p the single-threaded* $\texttt{VERIFY}(p)$ *is correct with respect to the validity of the* $\mathscr{N}\mathscr{D}$ *proof corresponding to p.*

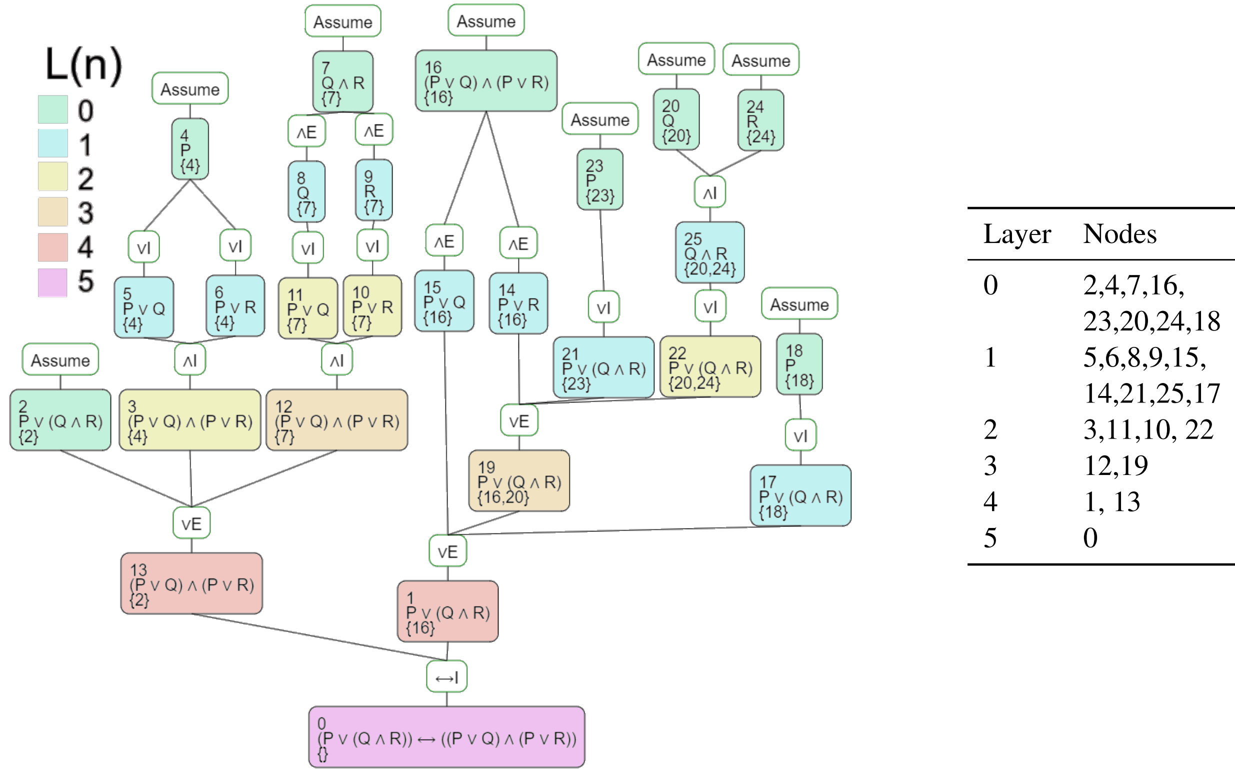
Proof. An algorithm is *correct* if it is sound and complete (termination is trivial). We prove completeness and soundness follows from symmetry. A natural deduction proof is valid if all steps are valid. For a step to be valid it needs to pass the *syntactic transformation* and the *assumption constraints*. From these, only the assumption constraint check requires information from outside the node and its parents. The rules of natural deduction in Figure 1 show how assumptions are computed based on the parent node's assumption sets. As such, parents of a node must be verified beforehand and have their assumptions computed. Nodes that are justified via assumptions mark the base case of this procedure as their assumption set only contains itself. Due to the definition of the layering in Equation 1 and its usage in Line 3, assumptions are in the first layer and the parents of a node must be in the previous layer. This means that the parents are verified and their assumptions are computed beforehand on lines 6-9. Inductively this means that all nodes are verified and have their assumptions computed successfully. Hence, the hypergraph proof itself is verified. □

3.2 Parallel Implementation (Non-optimized Parallel)

Within a layer, each node only depends on nodes in layers prior. This means that a node on some layer n does not depend on any other node on layer n. For our initial parallel implementation, fully described in Algorithm 2, we take advantage of this and verify the validity of each node on the same layer in parallel. In our implementation, we used a shared memory approach. To combat thread-safety issues we introduce a vector for each given layer called `aIds`. This vector lives in shared memory. Each entry holds a set of node ids and its length is the number of nodes in the current layer. After computing the layers, each node gets evenly distributed to the available threads. Each individual thread would then verify the nodes it was assigned and update the assumptions of the appropriate index within `aIds`. This is thread-safe because each entry only gets written to by the thread that its corresponding node id has been assigned to. The parallel portion of the algorithm additionally performs an `AND` reduction on the verification result of each node. This means that if any result of the individual verifications is false, then the entire verification result is false. After the parallel portion is finished, the global `assumption` map gets updated using `aIds`.

Let us illustrate the distribution of nodes with the example from Figure 5. In the graphic, the number at the top of each node represents its identifier. For brevity, we will use those numbers when referring to the nodes. As the zeroth layer only contains assumptions that are trivially verifiable, we will start our discussion from layer one. In this layer, we have the following nodes:

$$L(1) = \{5, 6, 8, 9, 15, 14, 21, 25, 17\}$$



Layer	Nodes
0	2,4,7,16, 23,20,24,18
1	5,6,8,9,15, 14,21,25,17
2	3,11,10, 22
3	12,19
4	1, 13
5	0

Figure 5: (Left) A proof of logical or ($\vee$) distributivity over logical and ($\wedge$). (Right) The nodes of the left proof grouped by layer.

For the sake of example, let us say we have three threads. Then thread 0 would be assigned $\{5,6,8\}$, thread 1 would be assigned $\{9,15,14\}$, and thread 2 would be assigned $\{21,25,17\}$. Let us focus on thread 0. Each of the three nodes verifies and the following assignments are made to `aIds` based on the justification used:

$$aIds(5) = \{assumptions(parent(5))\}$$
$$aIds(6) = \{assumptions(parent(6))\}$$
$$aIds(8) = \{assumptions(parent(8))\}$$

After the parallel portion, the data within `aIds` are copied into `assumptions` as a way to ensure thread-safety. On the next layer, we have the following nodes: $L(2) = \{3,11,10,22\}$. This then gets distributed with thread 0 getting $\{3,11\}$, thread 1 obtaining $\{10\}$, and thread 2 obtaining $\{22\}$. Focusing on the first thread again, the nodes verify and the following assignments are made to `aIds`:

$$aIds(3) = \{assumptions(5), assumptions(6)\}$$
$$aIds(11) = \{assumptions(8)\}$$

Notice that each of the items in those sets contains assumptions that were computed in the previous layer. They were originally assigned within `aIds` but then copied to `assumptions`. The results of each individual node verification were stored in `NodeValid` which is then AND-reduced into the variable `LayerValid`. Hence after the end of the parallel portion, the variable `LayerValid` would be true unless one of the nodes in the parallel portion failed to verify.

Algorithm 2 Multi-Threaded Algorithm

```
1: procedure VERIFY(ProofGraph p)
2:     Initialize assumptions to be empty.
3:     Create set of nodes on each layer using Equation 1 and store in layerMap.
4:     for layerNodes in layerMap do
5:         nl = length(layerNodes)
6:         aIds = sharedVector(length=nl)
7:         LayerValid = true
8:         for n in layerNodes in parallel do
9:             justification = just(n)
10:            ruleInfo = (m, assumptions(m)) ∀ m ∈ parents(n)
11:            NodeValid = is_valid(n, justification, ruleInfo)
12:            if NodeValid then
13:                Update aIds(n) using justification and ruleInfo
14:            AND_reduce(LayerValid, NodeValid)
15:        if not LayerValid then
16:            return false
17:        Update assumptions(n) using aIds
18:    return true
```

3.3 Multi-Threaded Static Load Balancing Optimization (Load Balancing)

Notice when going over Figure 5 in the last example that the distribution of nodes in the second layer was uneven. The assignment had one thread verifying two while the others verifying only a single one. In fact, we can speak to this more generally. Let T represent the number of threads available and l_i be the number of nodes in layer i. Let us assume that the verification of one node takes two units of work total: one to verify the syntax and one to verify the assumptions. Let $m = l_i \bmod T$. Then, if $m \neq 0$ there are $(T - m)$ threads that are doing one less unit of work.

It is with this consideration that we look at static load balancing, presented in Algorithm 3. For the threads with one less unit of work, they take a node from a future layer and syntax verify them. This approach is valid because syntax verification only requires the current node and its parents' formulae which are stored in the proof graph and does not require additional information from the prior layers such as assumption sets. In order to ensure that the nodes that are syntax verified by the remaining $(T - m)$ threads are distinct, we make use of another reduction variable `numSyntaxVerified`. Each thread would be assigned the node that's the sum of that variable and its thread id. Do note that this is different from dynamic load balancing as the amount of work is evenly distributed and does not take into account during runtime some threads finishing before others.

Let us turn to our example from Figure 5 again. Recall that the distribution of work at layer two was the following: thread 0 maps to $\{3, 11\}$, thread 1 has $\{10\}$, and thread 2 has $\{22\}$. We can then squeeze in syntax verification checks in thread 1 and thread 2. Then, the new allocation becomes: thread 0 maps to $\{3, 11\}$, thread 1 maps to $\{10, x_s\}$, and thread 2 maps to $\{22, y_s\}$. The subscript denotes how we are only performing a syntax verification at that step. Recall that we can not perform full verification of nodes in future layers because we do not know if there's a node on the current layer that its assumptions depends on. The question then is: how are x_s and y_s calculated? As noted before, this is where we keep track of the total number of nodes that we have syntax verified already. If we have a flat vector of

Algorithm 3 Multi-Threaded Load Balance Algorithm

```
1:  procedure VERIFY(ProofGraph p)
2:      Initialize assumptions to be empty.
3:      Create set of nodes on each layer using Equation 1 and store in layerMap.
4:      AllNodes = Flatten(layers)
5:      numSyntaxVerified = 0
6:      for layerNodes in layerMap do
7:          nl = length(layerNodes)
8:          aIds = sharedVector(length=nl)
9:          LayerValid = true
10:         layerSyntaxVerified = 0
11:         for n in layerNodes in parallel do
12:             justification = just(n)
13:             ruleInfo = (m, assumptions(m)) ∀ m ∈ parents(n)
14:             NodeValid = is_valid(n, ruleInfo)
15:             if NodeValid then
16:                 Update aIds(n) using justification and ruleInfo
17:             threadIterSyntaxVerified = 1
18:             if thread verifying less nodes then
19:                 extraN = AllNodes[numSyntaxVerified + threadId]
20:                 NodeValid = NodeValid and syntaxVerify(extraN, parents(extraN))
21:                 threadIterSyntaxVerified = 2
22:             SUM_reduce(layerSyntaxVerified, threadIterSyntaxVerified)
23:             AND_reduce(LayerValid, NodeValid)
24:         if not LayerValid then
25:             return false
26:         Update assumptions(n) using aIds
27:         numSyntaxVerified = numSyntaxVerified + layerSyntaxVerified
28:     return true
```

all nodes that are partially ordered by their layer number, then for thread i we can have it syntax verify `numSyntaxVerified`$+i$ element of that flat vector. A sum reduction then keeps track of the total number of syntax verifications performed on a given layer which is later used to update `numSyntaxVerified`.

3.4 Parallel Distribution of Syntax Checks (Syntax-First)

In the last section we discussed that syntax verification can occur beyond the current layer being considered. In fact, syntax verification can occur outside of the layering structure in general which this optimization considers. In this approach we first perform the syntax verification in parallel over all nodes before iterating over the layers. This approach is outlined in Algorithm 4. This not only has the benefit of lowering the time to find a syntactic error, but also more evenly distributes the syntax verification over all threads.

For our example in Figure 5, the proof graph contains node ids 0 through 25. If we have three threads, then thread 0 would be assigned nodes 0 through 8, thread 1 would be assigned nodes 9 through 16, and

thread 2 would be assigned nodes 17 through 25. Each thread would then loop over their assigned nodes and syntax verify them. When the three threads finish, if any of their nodes failed to syntax verify, then the algorithm would end and the proof graph verification would fail. In this example, however, the nodes pass the syntax verification. The rest of the algorithm closely follows Algorithm 2 where instead of performing a full verification, we only verify that the assumption constraints hold.

Algorithm 4 Multi-Threaded Syntax Check First Algorithm

```
 1: procedure VERIFY(ProofGraph p)
 2:     syntaxValid = True
 3:     for n in p.nodes in parallel do
 4:         valid = verifySyntax(n, parents(n))
 5:         AND_reduce(syntaxValid, valid)
 6:     if not syntaxValid then
 7:         return false
 8:     Initialize assumptions to be empty.
 9:     Create set of nodes on each layer using Equation 1 and store in layerMap.
10:     for layerNodes in layerMap do
11:         nl = length(layerNodes)
12:         aIds = sharedVector(length=nl)
13:         LayerValid = true
14:         for n in layerNodes in parallel do
15:             justification = just(n)
16:             ruleInfo = (m, assumptions(m)) ∀ m ∈ parents(n)
17:             NodeValid = verifyAssumptions(n, justification, ruleInfo)
18:             if NodeValid then
19:                 Update aIds(n) using justification and ruleInfo
20:             AND_reduce(LayerValid, NodeValid)
21:         if not LayerValid then
22:             return false
23:         Update assumptions(n) using aids
24:     return true
```

4 Methodology and Results

To discuss the performance of our algorithms, we provide an empirical investigation. To this end, we look toward a comparison of the number of seconds needed to verify various proof structures using the algorithms described before. Inspired by the topologies used in computer network design [2], we introduce a directed variant that we call directed acyclic network topologies or DANTs. These DANTs represent different classes of possible proofs with which we perform benchmarks over.

4.1 Directed Acyclic Network Topologies (DANTs)

We have identified three distinct classes of DANTs which we include in our benchmark of proof graphs for analysis. The pictorial depiction is shown in Figure 6 and is generated as so: **Straight Line** (n):

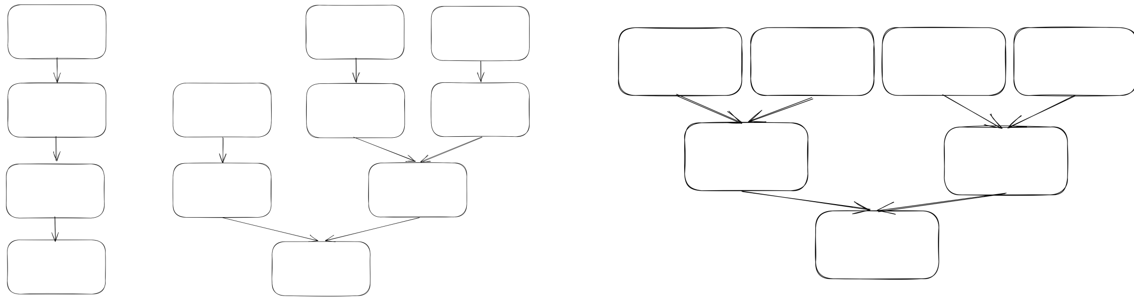


Figure 6: Straight, Parallel-Branch, then Tree Topologies

Parameterized by the total number of nodes n, this topology only contains one assumption at the top and then each future inference is a disjunction introduction. This enforces a straight linear proof with no branching. We do not expect any speedup in the parallel algorithm in this case as each layer only has one node. **Parallel Branches** (b,n): This topology emulates multiple lines of independent reasoning before combining towards the end. It starts off with b separate assumptions and then performs a disjunctive introduction on each assumption n times before iteratively applying conjunctive introduction to each branch until there is one remaining. We expect the number of branches will correlate with the scalability of verification. It should be noted that this topology is isomorphic to a straight topology of length n when $b = 1$ and emulates a tree like topology of height b when $b > 1$. **Tree** (h): In this topology we generate 2^h assumptions and iteratively apply conjunction introduction h times until we reach a single node. This creates a balanced binary tree. We hypothesize the greatest amount of speedup from this topology.

4.2 Empirical Analysis

We perform two classes of experiments: (1) a parallel strong scaling study in which the proof to be verified is held constant while the number of processors increases; (2) a problem size scaling study in which we hold the number of threads constant and look at how each method performs as the problem gets harder.

4.2.1 Implementation Details

Our benchmarks were performed on one node of the IBM DCS supercomputer, AiMOS, at Rensselaer Polytechnic Institute. Our code is available at `https://github.com/RAIRLab/Parallel-Verifier` [3]. The code is implemented in C++11 and makes use of the standard C++ library data structures. For the multiprocessing component, we use the OpenMP library [5]. OpenMP operates over software threads which are assigned to CPUs. We ensure during our scaling study that the system is not *oversubscribed*, meaning that there is just a single thread used per CPU. AiMOS provides us a single node on which ten physical cores are available; however one is reserved for the operating system and IO, therefore nine are used for our experiments. For our benchmarks we do not include the time it took for initialization, file parsing, or proof parsing; we only measure the time taken to verify the proof. For this, we record the clock-cycles before and after the execution of the verification algorithm and use their difference to compute the total cycles. We then compute the number of seconds taken by each method through dividing the number of total cycles by the base clock rate of 512MHz.

[3] For reproducibility of our results, please see the following link for the specific commit the results of this paper is based on: `https://github.com/RAIRLab/Parallel-Verifier/tree/a661abbe5bf038a3fa8645b8af532b0a60daebe5`

4.2.2 Strong Scaling

For our strong scaling study we vary the number of threads used while holding the DANT instance constant. For the straight topology, we consider a length n of 150. For the branch topology, we consider $b = 150$ branches each with a length of $n = 100$. Lastly, for the tree topology, we consider $h = 16$ conjunction introductions for a total number of 2^{16} vertices. Results can be seen on the left of Figure 7. The strong scaling results show a clear benefit to our parallelized verification approach. In the case of the straight topology the serial algorithm vastly outperforms the parallel algorithms, which is expected as in this topology there is only one node per layer. There are clear overheads to parallelization, such as waiting for all threads to finish, that make timing differences visible as the number of threads increases for the straight topology. For the branch topology with 150 branches we see that our parallel methods scale well, particularly load balancing which beats out syntax-first and non-optimized parallel methods. We hypothesize this is due to the number of remaining nodes on each layer remaining constant which allows for a good balance of syntax checking vs assumption updating. The parallel methods perform quite well on the tree topology significantly beating out the serial method, with non-optimized parallel and syntax-first methods beating out load balancing likely due to the overhead costs.

4.2.3 Problem Scaling

For our problem scaling study we hold the number of threads constant (at AiMOS' maximum value of nine) and vary the problem size. For the straight topology, we consider a chain of disjunctive introductions of lengths (n) 100, 150, 200, 250, 300, 350, and 400. For the branch topology, we consider a fixed branch length of $n = 100$ and vary the number of branches (b) at 30, 50, 70, 90, 110, 130, and 150. Lastly for the tree topology, we create binary trees of heights (h) 8, 10, 12, 14, 16, 18, and 20. Results can be seen on the right side of Figure 7. We hypothesize the straight topology scaling is not linear due to overheads such as the formulae length increasing as the problem size increases. We see that for all problem sizes on the straight topology, the serial implementation outperforms the parallel implementation. This aligns with the observation in the strong scaling study that the parallel methods have overheads and the fact that for all parallel methods, only one node is on each layer, preventing the majority of the threads from doing any work. In the branch topology, the results show that as the number of branches increases, the effectiveness of parallel methods increases. This is particularly shown in load balancing, due to the reasons discussed in §4.2.2. For the tree topology, there is an exponential increase in the time taken as the problem grows, largely due to the fact that the number of nodes to verify increases exponentially (2^n) as the problem size increases. We see that as the problem size grows, the performance of our parallel methods over the serial method increases substantially.

5 Related Work

Past work has investigated parallel or concurrent verification of other logical calculi. For example the developers of Isabelle [24], a proof assistant with support for first-order logic, higher-order logic, and Zermelo-Fraenkel set theory, used concurrent programming for the efficient verification of LCF-style proofs [18]. In their work, during the verification of a proof, if a reference to another proof is made, and that proof has yet to be verified, then a promise is deferred. At the end of verification, all promises are resolved and localized errors are then shown to the user [21, 22]. This work operates over entire proofs while we focus on parallelizing steps within a proof. In [23], Wenzel introduces what he calls granularities of concurrency within verifying a single proof. The levels include concurrent verification

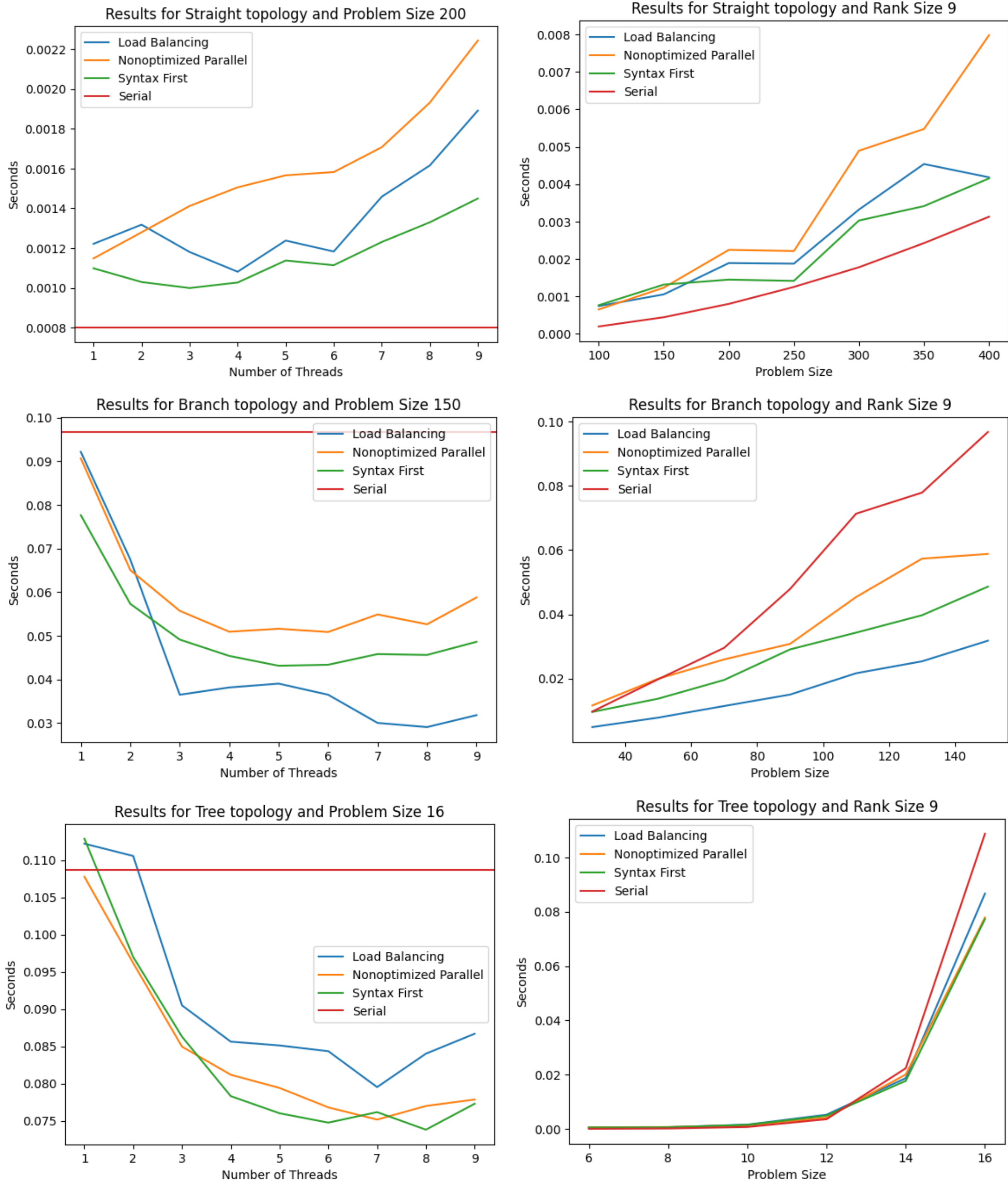


Figure 7: (Left) Strong scaling study results (Right) Problem scaling study results.

of theories, concurrent verification of commands, and concurrent verification of subproofs. In terms of our work, we do not consider extra background theories, commands correspond to our proof steps, and we do not consider subproofs in our work. As discussed in §2, there is a definition of a subproof in our natural deduction hypergraphs. However, it's not something specified by the creator of the proof and there can be n different subproofs for a proof with n nodes.

Färber looked at concurrent verification of commands in his work parallelizing proof checking inside the lambda-Pi calculus modulo rewriting [8]. In this work, he breaks up a command into four tasks: parsing, sharing, type inference, and type checking. Similarly, our work breaks up our inference rules into two steps: syntactic checks and assumption checks. We additionally look at the parallel verification of sets of steps or commands, as opposed to only looking at the concurrency within each command.

6 Conclusion

In this work, we presented a layering based algorithm that decouples the underlying semantic dependencies of proof steps in natural deduction. Through this, we introduced a suite of new algorithms which use layers to parallelize verification of hypergraphical natural deduction proofs. Directed acyclic network topologies (DANTs) were introduced as a benchmark for hypergraphical proofs and we have shown in our analysis that the parallel algorithms perform better than their serial counterpart on non-straight DANT instances. These parallel algorithms were additionally shown to scale through both the strong scaling and problem scaling studies. This work has applications in formal verification, specifically in proof assistants.

Our future work falls into four categories: theoretical results, empirical results, logic extensions, algorithmic optimizations: (1) For theoretical results, we would benefit from analysis with respect to Amdahl's Law [11] to calculate the overall speedup with respect to different parallelizable tasks in each of the algorithms. (2) For further empirical results, we can test randomized proof topologies or craft a dataset of common natural deduction proofs. (3) We wish to extend our verifier to handle different types of logics, specifically first-order and modal logics. First-order logics are used heavily within proof assistants, and require the ability to represent and reason over formulae at the term level, including the need for checking if variables are free or bound in inference rules. We conjecture that despite these extra requirements, our layer based parallel approaches would still work on first-order proof graphs. In order to handle modal logics such as **K5**, we would have to adapt the algorithm to include additional bookkeeping required for several of the inference rules. (4) We would like to explore approaches to scale beyond a single computer. This involves exploring message-passing parallelism which is often used in distributed computing.

Acknowledgements This paper was supported in part by a fellowship award under contract FA9550-21-F-0003 through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR) and the Army Research Office (ARO).

References

[1] Konstantine Arkoudas & Selmer Bringsjord (2009): *Vivid: A framework for combining diagrammatic and symbolic reasoning*. Artificial Intelligence 173(15), pp. 1367–1405, doi:10.1016/j.artint.2009.06.002.

[2] B Bicsi (2002): *Network design basics for cabling professionals*. City: McGraw-Hill Professional, pp. 0885–8950.

[3] Selmer Bringsjord (2021): *Intermediate Formal Logic and AI*.

[4] Selmer Bringsjord, Naveen Sundar Govindarajulu, Joshua Taylor & Alexander Bringsjord (2022): *Logic: A Modern Approach*. Motalen.

[5] Rohit Chandra, Leo Dagum, Ramesh Menon, David Kohr, Dror Maydan & Jeff McDonald (2001): *Parallel programming in OpenMP*. Morgan kaufmann.

[6] The mathlib Community (2019): *The lean mathematical library*. Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, doi:10.1145/3372885.3373824.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein (2009): *Introduction to Algorithms*, 2 edition. MIT Press. Available at `https://api.semanticscholar.org/CorpusID:60621753`.

[8] Michael Färber (2022): *Safe, fast, concurrent proof checking for the lambda-pi calculus modulo rewriting*. Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, doi:10.1145/3497775.3503683.

[9] Frederic Brenton Fitch (1953): *Symbolic Logic, An Introduction*. American Journal of Physics 21, pp. 237–237, doi:10.2307/2020641.

[10] Gerhard Gentzen (1935): *Untersuchungen über das logische Schließen. I*. Mathematische Zeitschrift 39, pp. 176–210, doi:10.1007/bf01201353.

[11] Mark D. Hill & Michael R. Marty (2008): *Amdahl's Law in the Multicore Era*. IEEE Computer 41, pp. 33–38, doi:10.1109/MC.2008.209.

[12] Stanisław Jaśkowski (1934): *On the rules of suppositions in formal logic*. Studia Logica.

[13] Vipin Kumar, Ananth Y. Grama, Anshul Gupta & George Karypis (1994): *Introduction to parallel computing: design and analysis of algorithms*. Benjamin-Cummings Publishing Co., Inc.

[14] Leonardo de Moura & Sebastian Ullrich (2021): *The Lean 4 Theorem Prover and Programming Language*. In: International Conference on Automated Deduction, Springer, pp. 625–635, doi:10.1007/978-3-030-79876-5_37.

[15] James Oswald & Brandon Rozek (2022): *Lazyslate*. Available at `https://github.com/James-Oswald/lazyslate`.

[16] Francis Jeffry Pelletier & Allen Hazen (2023): *Natural Deduction Systems in Logic*. In Edward N. Zalta & Uri Nodelman, editors: The Stanford Encyclopedia of Philosophy, Spring 2023 edition, Metaphysics Research Lab, Stanford University, pp. 1–1.

[17] Dag Prawitz (1965): *Natural Deduction: A Proof-Theoretical Study*. Dover Publications, doi:10.2307/2271676.

[18] Dana S Scott (1993): *A type-theoretical alternative to ISWIM, CUCH, OWHY*. Theoretical Computer Science 121(1-2), pp. 411–440, doi:10.1016/0304-3975(93)90095-B.

[19] The Coq Development Team (2019): *The Coq Proof Assistant*, doi:10.5281/zenodo.3476303.

[20] Vitaly I. Voloshin (2013): *Introduction to Graph and Hypergraph Theory*. Nova Kroshka Books.

[21] Makarius Wenzel (2009): *Parallel Proof Checking in Isabelle/Isar*.

[22] Makarius Wenzel (2013): *READ-EVAL-PRINT in parallel and asynchronous proof-checking*. arXiv preprint arXiv:1307.1944, doi:10.4204/eptcs.118.4.

[23] Makarius Wenzel (2013): *Shared-Memory Multiprocessing for Interactive Theorem Proving*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: Interactive Theorem Proving, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 418–434, doi:10.1007/978-3-642-39634-2_30.

[24] Makarius Wenzel, Lawrence C Paulson & Tobias Nipkow (2008): *The isabelle framework*. In: International Conference on Theorem Proving in Higher Order Logics, Springer, pp. 33–38, doi:10.1007/978-3-540-71067-7_7.

An Interpretation of $\mathbf{E}\text{-}\mathbf{HA}^{\omega}$ inside $\mathbf{HA}^{\omega}$

Félix Castro

IRIF
Université Paris Cité
Paris, France

IMERL
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay

castro@irif.fr

Higher Type Arithmetic ($\mathbf{HA}^{\omega}$) is a first-order many-sorted theory. It is a conservative extension of Heyting Arithmetic obtained by extending the syntax of terms to all of System T: the objects of interest here are the functionals of “higher types”. While equality between natural numbers is specified by the axioms of Peano, how can equality between functionals be defined? From this question, different versions of $\mathbf{HA}^{\omega}$ arise, such as an extensional version ($\mathbf{E}\text{-}\mathbf{HA}^{\omega}$) and an intentional version ($\mathbf{I}\text{-}\mathbf{HA}^{\omega}$). In this work, we will see how the study of partial equivalence relations leads us to design a translation by parametricity from $\mathbf{E}\text{-}\mathbf{HA}^{\omega}$ to $\mathbf{HA}^{\omega}$.

1 Introduction

In second-order logic, it can be shown as a meta-theorem that two extensionally equal predicates satisfy the same properties. It is not the case in higher-order logic: this is due to the potential existence of non-extensional (higher-order) predicates. However, Gandy showed that axioms of extensionality could be consistently added by restraining the range of quantification to *extensional* elements [4]. A similar phenomenon occurs in Higher Type Arithmetic ($\mathbf{HA}^{\omega}$): one cannot prove in $\mathbf{HA}^{\omega}$ that two extensionally equal functions satisfy the same formulas. It can be seen for instance by working in the model of Hereditary Recursive Operations **HRO** [10] where a functional can inspect the source code of its argument. But, again, axioms of extensionality can be added without loss of consistency: Zucker showed that every model of $\mathbf{N}\text{-}\mathbf{HA}^{\omega}$ (Higher Type Arithmetic with equality at all levels of sort) can be turned into a model of $\mathbf{E}\text{-}\mathbf{HA}^{\omega}$ (Higher Type Arithmetic with extensional equality at all levels of sort) [12].

In this work we tackle a similar problem. Starting from $\mathbf{HA}^{\omega}$, we show that an extensional equality can be consistently added at all levels of sorts. Taking inspiration from syntactical models of type theory [3, 1], we choose to do it in a syntactical fashion: we design an interpretation of $\mathbf{E}\text{-}\mathbf{HA}^{\omega}$ in $\mathbf{HA}^{\omega}$ that we express as a translation between two proof systems (without reduction rules). Concretely, we will compile a language with extensional equality at all levels of sorts to a language that merely has equality in the sort $\mathbf{N}$. It will be done using techniques of parametricity, as one goal of this paper is to emphasize that parametricity can be used to extend equality.

After exposing a proof system $\lambda\mathbf{HA}^{\omega}$ that captures Higher Type Arithmetic (Section 2), we will study families (indexed by the sorts of System T) of (internal) partial equivalence relations that could be used to extend equality (Section 3). In particular, we will compare two potential candidates:

1. a family $=^{\text{ext}}_{\sigma}$ generated from equality (over $\mathbf{N}$) in an extensional fashion;
2. a family $=^{\text{pm}}_{\sigma}$ generated from equality (over $\mathbf{N}$) in a way reminiscent of binary parametricity [9].

A. Ciaffaglione and C. Olarte (Eds.): Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2023)
EPTCS 396, 2023, pp. 52–66, doi:10.4204/EPTCS.396.5

© F. Castro
This work is licensed under the Creative Commons Attribution License.

$$\frac{}{\Delta_1, x^\sigma, \Delta_2 \vdash_T x^\sigma : \sigma}$$

$$\frac{\Delta, x^\sigma \vdash_T t : \tau}{\Delta \vdash_T \lambda x^\sigma . t : \sigma \to \tau} \qquad \frac{\Delta \vdash_T t : \sigma \to \tau \quad \Delta \vdash_T u : \sigma}{\Delta \vdash_T tu : \tau}$$

$$\frac{}{\Delta \vdash_T 0 : \mathbf{N}} \qquad \frac{\Delta \vdash_T t : \mathbf{N}}{\Delta \vdash_T \mathrm{S}\, t : \mathbf{N}}$$

$$\frac{\Delta \vdash_T t : \sigma \quad \Delta \vdash_T u : \sigma \to \mathbf{N} \to \sigma \quad \Delta \vdash_T v : \mathbf{N}}{\Delta \vdash_T \mathrm{Rec}^\sigma\, t\, u\, v : \sigma}$$

Figure 1: Derivation in System T

While the former is reflexive, the latter is not. But being reflexive is not desirable in this context. Indeed, as explained above, one needs to restrict the range of quantifications before extending equality: specifically we will restrict quantifications on a sort σ to the domain of $=^{\mathrm{pm}}_{\sigma}$. Our first translation will be used to show that each closed term of System T is indeed in the domain of $=^{\mathrm{pm}}$: we translate judgments of System T into judgments of $\lambda\mathbf{HA}^\omega$ and we follow the typical translation by parametrecity, as it will allow us to show that typed terms satisfy the relation linked to their type [9, 11, 2]. Finally, by keeping the idea of a translation by parametricity, we will translate a proof system $\lambda\mathbf{E}\text{-}\mathbf{HA}^\omega$ (capturing $\mathbf{E}\text{-}\mathbf{HA}^\omega$) to $\lambda\mathbf{HA}^\omega$ (Section 4). Before concluding, we will compare our result and our methodology to related work (Section 5).

2 A proof system for Higher Type Arithmetic

2.1 System T

We use a version of Gödel's System T obtained by extending the simply typed λ-calculus (à la Church) with a type constant $\mathbf{N}$ and native constructors to use it. Terms, sorts and signatures of System T are described as follows:

$$\begin{array}{llll} \text{Sorts} & \sigma, \tau & ::= & \mathbf{N} \mid \sigma \to \tau \\ \text{Terms} & t, u & ::= & x^\sigma \mid \lambda x^\sigma . t \mid tu \\ & & & \mid 0 \mid \mathrm{S}\, t \mid \mathrm{Rec}^\sigma\, t\, u\, v \\ \text{Signatures} & \Delta & ::= & \emptyset \mid \Delta, x^\sigma \end{array}$$

System T is presented in Church's style so terms come associated with a unique sort. Nevertheless, we use a type system (see Figure 1 page 53) to take into account in which signature (or environment) a term is considered. We may omit sort annotations on variables.

We consider the following rules on terms

$$\begin{array}{rcl} (\lambda x.t)\, u & \succ & t[x ::= u] \\ \mathrm{Rec}\, t\, u\, 0 & \succ & t \\ \mathrm{Rec}\, t\, u\, (\mathrm{S}\, v) & \succ & u\, (\mathrm{Rec}\, t\, u\, v)\, v \end{array}$$

from which we generate reduction and congruence $t \rightsquigarrow u$ and $t \cong u$ as respectively the least reflexive, transitive and closed by congruence relation containing $\succ$ and the least closed by congruence equivalence relation containing $\succ$. We define a substitution θ to be a finite function from variables to terms. The action of a substitution θ on terms, denoted $t[\theta]$, corresponds to the simultaneous substitutions of free variables x in the domain of θ by $\theta(x)$.

Metatheoretical results about System T can be found in the book of Girard, Lafont and Taylor [5], for instance:

1. terms of System T are strongly normalizable;
2. closed normal terms of type **N** are of the form $\mathrm{S}^n 0$, closed normal terms of type $\sigma \to \tau$ are of the form $\lambda x^\sigma.t$.

Finally, we will use the two following facts.

Fact 1. *A generalized version of the weakening rule is admissible for this system:*

$$\text{if} \quad \Delta \subseteq \Delta' \text{ and } \Delta \vdash_{\mathcal{T}} t : \sigma \quad \text{then} \quad \Delta' \vdash_{\mathcal{T}} t : \sigma$$

where $\Delta \subseteq \Delta'$ is interpreted as the set-theoretic inclusion (while seeing signatures as sets).

Fact 2. *If θ is a substitution then $t \cong u$ implies $t[\theta] \cong u[\theta]$.*

2.2 Higher Type Arithmetic

Higher Type Arithmetic ($\mathbf{HA}^\omega$) is a theory of many-sorted first order logic. It is a conservative extension of **HA** obtained by extending the term language to the System T. Models of $\mathbf{HA}^\omega$ are described in the book of Troelstra [10], in particular the following will be used in the sequel:

1. the set-theoretic model **M** defined by

$$\begin{array}{rcl} \mathbf{M}_{\mathbf{N}} & \equiv & \mathbb{N} \\ \mathbf{M}_{\sigma\to\tau} & \equiv & \mathbf{M}_\tau^{\mathbf{M}_\sigma} \end{array}$$

2. the model of Hereditary Recursive Operations **HRO** defined by

$$\begin{array}{rcl} \mathbf{HRO}_{\mathbf{N}} & \equiv & \mathbb{N} \\ \mathbf{HRO}_{\sigma\to\tau} & \equiv & \{e \in \mathbb{N} \mid \forall n \in \mathbf{HRO}_\sigma \ \{e\}(n) \downarrow\in \mathbf{HRO}_\tau\} \end{array}$$

   where $\{e\}(n) \downarrow\in E$ means that the computation of the function of index e terminates on the input n and that the result of this computation is in E.

We define a proof system $\lambda\mathbf{HA}^\omega$ that captures $\mathbf{HA}^\omega$. Formulas, proof terms and contexts of $\lambda\mathbf{HA}^\omega$ are generated by the following grammar:

$$\begin{array}{llcl} \text{Formulas} & \Phi,\Psi & ::= & t = u \mid \bot \mid \mathrm{null}(t) \\ & & & \mid \Phi \Rightarrow \Psi \mid \Phi \wedge \Psi \\ & & & \mid \forall x^\sigma \Phi \mid \exists x^\sigma \Phi \\ \text{Proof terms} & M,N & ::= & \xi \mid \mathrm{refl}\, t \mid \mathrm{peel}^{t,u}(M, \hat{x}.\Phi, N) \mid \mathrm{efq}(M,\Phi) \\ & & & \mid \lambda\xi.M \mid M\,N \\ & & & \mid (M,N) \mid M.1 \mid M.2 \\ & & & \mid \lambda x^\sigma.M \mid M\,t \\ & & & \mid [t,M] \mid \mathrm{let}\ [x,\xi] := M \ \mathrm{in}\ N \\ & & & \mid \mathrm{Ind}(\hat{x}.\Phi, M, N, t) \\ \text{Contexts} & \Gamma & ::= & \emptyset \mid \Gamma, \xi : \Phi \end{array}$$

$$\frac{(\Delta;\Gamma)\ \mathbf{wf}}{\Delta;\Gamma\vdash \xi:\Phi}\,(\xi:\phi\in\Gamma)\qquad \frac{\Delta;\Gamma\vdash M:\bot}{\Delta;\Gamma\vdash \mathrm{efq}(M,\Phi):\Phi}\,(\mathbf{FV}(\Phi)\subseteq\Delta)\qquad \frac{\Delta;\Gamma\vdash M:\Phi}{\Delta;\Gamma\vdash M:\Psi}\,(\Phi\simeq\Psi)$$

$$\frac{\Delta;\Gamma,\xi:\Phi\vdash M:\Psi}{\Delta;\Gamma\vdash \lambda\xi.M:\Phi\Rightarrow\Psi}\qquad \frac{\Delta;\Gamma\vdash M:\Phi\Rightarrow\Psi\quad \Delta;\Gamma\vdash N:\Phi}{\Delta;\Gamma\vdash MN:\Psi}$$

$$\frac{\Delta;\Gamma\vdash M_1:\Phi_1\quad \Delta;\Gamma\vdash M_2:\Phi_2}{\Delta;\Gamma\vdash (M_1,M_2):\Phi_1\wedge\Phi_2}\qquad \frac{\Delta;\Gamma\vdash M:\Phi_1\wedge\Phi_2}{\Delta;\Gamma\vdash M.i:\Phi_i}\,(i=1,2)$$

$$\frac{\Delta,x^\sigma;\Gamma\vdash M:\Phi}{\Delta;\Gamma\vdash \lambda x^\sigma.M:\forall x^\sigma\Phi}\,(x^\sigma\notin\mathbf{FV}(\Gamma))\qquad \frac{\Delta;\Gamma\vdash M:\forall x^\sigma\Phi\quad \Delta\vdash_{\mathrm{T}} t:\sigma}{\Delta;\Gamma\vdash Mt:\Phi[x^\sigma:=t]}$$

$$\frac{\Delta;\Gamma\vdash M:\Phi[x^\sigma:=t]\quad \Delta\vdash_{\mathrm{T}} t:\sigma}{\Delta;\Gamma\vdash [t,M]:\exists x^\sigma\Phi}\qquad \frac{\Delta;\Gamma\vdash M:\exists x^\sigma\Phi\quad \Delta,x^\sigma;\Gamma,\xi:\Phi\vdash N:\Psi}{\Delta;\Gamma\vdash \mathrm{let}\ [x,\xi]:=M\ \mathrm{in}\ N:\Psi}\,(x^\sigma\notin\mathbf{FV}(\Gamma,\Psi))$$

$$\frac{(\Delta;\Gamma)\ \mathbf{wf}\quad \Delta\vdash_{\mathrm{T}} t:\mathbf{N}}{\Delta;\Gamma\vdash \mathrm{refl}\,t:t=t}\qquad \frac{\Delta;\Gamma\vdash M:t=u\quad \Delta;\Gamma\vdash N:\Phi[x^{\mathbf{N}}:=t]}{\Delta;\Gamma\vdash \mathrm{peel}^{t,u}(M,\hat{x}.\Phi,N):\Phi[x^{\mathbf{N}}:=u]}$$

$$\frac{\Delta;\Gamma\vdash M:\Phi[x^{\mathbf{N}}:=0]\quad \Delta;\Gamma\vdash N:\forall x^{\mathbf{N}}.(\Phi\Rightarrow\Phi[x^{\mathbf{N}}:=\mathrm{S}x^{\mathbf{N}}])\quad \Delta\vdash_{\mathrm{T}} t:\mathbf{N}}{\Delta;\Gamma\vdash \mathrm{Ind}(\hat{x}.\Phi,M,N,t):\Phi[x:=t]}$$

Figure 2: Proof derivations in $\lambda\mathbf{HA}^\omega$

This syntax contains three different λ-abstractions: two λ-abstractions at the level of proof terms ($\lambda\xi.M$ and $\lambda x^\sigma.M$) and the λ-abstraction of System T at the level of formulas ($\lambda x^\sigma.t$). The sort annotation on a variable may be omitted in the sequel if it can be inferred. In the proof terms $\mathrm{peel}^{t,u}(M,\hat{x}.\Phi,N)$ and $\mathrm{Ind}(\hat{x}.\Phi,M,N,t)$, the variable x is bound in Φ: the binder $\hat{x}$ is used to specify which variable will be substituted. The connectives $\top$ and $\vee$ are not included in $\lambda\mathbf{HA}^\omega$ but can be defined as $\top\equiv\bot\Rightarrow\bot$ and $\Phi\vee\Psi\equiv\exists x^{\mathbf{N}}\,(x=0\Rightarrow\Phi\wedge x\neq 0\Rightarrow\Psi)$ where the relation $x\neq y$ denotes $x=y\Rightarrow\bot$.

We consider sequents of the form $\Delta;\Gamma\vdash M:\Phi$ where

1. Δ is a signature of System T;
2. Γ is a context of $\lambda\mathbf{HA}^\omega$.

The typing rules of $\lambda\mathbf{HA}^\omega$ are presented in Figure 2 at page 55. Note that equality is only defined on the sort $\mathbf{N}$. A pair of a signature and a context $(\Delta;\Gamma)$ is well formed when the free first-order variables of Γ are contained in Δ, i.e.

$$(\Delta;\Gamma)\ \mathbf{wf}\quad\equiv\quad\mathbf{FV}(\Gamma)\quad\subseteq\quad\Delta$$

This system is not equipped with reduction rules for proof terms: they are used as annotations for the derivation and they serve as a tool to formulate our work as a fully specified translation. The congruence relation $\Phi\simeq\Psi$ between formulas used in $\lambda\mathbf{HA}^\omega$ is generated from the reduction rules of System T and two extra rules:

$$\begin{array}{rcl}\mathrm{null}(0) & \succ & \top\\ \mathrm{null}(\mathrm{S}x) & \succ & \bot\end{array}$$

Because of the conversion rule, terms of System T are treated up to the equivalence $\cong$. For instance, one can prove $(\lambda x^{\mathbf{N}}.x)0=0$ in $\lambda\mathbf{HA}^\omega$. Moreover, all the axioms of $\mathbf{HA}^\omega$ [10] are derivable. In particular, the predicate $\mathrm{null}(t)$ is used to prove $\forall x^{\mathbf{N}}\ \mathrm{S}x\neq 0$.

Fact 3. *$\lambda\mathbf{HA}^\omega$ captures $\mathbf{HA}^\omega$ in the following manner: a closed formula Φ is derivable in $\lambda\mathbf{HA}^\omega$ if and only if the formula obtained from Φ by replacing all occurrences of subformulas* $\mathrm{null}(t)$ *by $t = 0$ is a logical consequence of $\mathbf{HA}^\omega$.*

The notion of substitutions extends from System T to $\lambda\mathbf{HA}^\omega$. Concretely, a (first-order) substitution θ is a finite function from first-order variables $(x, y...)$ to terms of System T, while the operation of substitution on proof terms M and formulas Φ is defined as before. The notation $\Gamma[\theta]$ represents the application of the substitution θ to all terms and formulas in the context Γ. The system $\lambda\mathbf{HA}^\omega$ satisfies the following properties:

Fact 4. *If $\Delta;\Gamma \vdash M : \Phi$ then $FV(\Phi) \subseteq \Delta$.*

Fact 5. *A generalized version of the weakening rule is admissible for this system:*

$$\textit{if} \quad \Delta \subseteq \Delta',\, \Gamma \subseteq \Gamma' \textit{ and } \Delta;\Gamma \vdash M : \Phi \quad \textit{then} \quad \Delta';\Gamma' \vdash M : \Phi$$

where the set-theoretic inclusion is used to compare signatures and contexts.

Fact 6. *Let θ be a substitution of first-order variables, Δ a signature included in its domain and Δ' a signature containing all free variables of its image. Then*

$$\Delta;\Gamma \vdash M : \Phi \quad \textit{implies} \quad \Delta';\Gamma[\theta] \vdash M[\theta] : \Phi[\theta]$$

Fact 7. *If $\Delta;\Gamma,\xi : \Psi \vdash M : \Phi$ and $\Delta;\Gamma \vdash N : \Psi$ then $\Delta;\Gamma \vdash M[\xi := N] : \Phi$.*

3 A preliminary study of possible extensions of equality

3.1 Two examples of Partial Equivalence Relation

Let σ be a sort of System T. A symbol of binary relation $\mathscr{R}$ on σ (added to the syntax of $\lambda\mathbf{HA}^\omega$) is a partial equivalence relation when it is symmetric and transitive. It is the case if the formulas

$$\begin{array}{rcl} \mathbf{Sym}_{\mathscr{R}} & \equiv & \forall x^\sigma \forall y^\sigma\ x\mathscr{R}y \Rightarrow y\mathscr{R}x \\ \mathbf{Trans}_{\mathscr{R}} & \equiv & \forall x^\sigma \forall y^\sigma \forall z^\sigma\ x\mathscr{R}y \Rightarrow y\mathscr{R}z \Rightarrow x\mathscr{R}z \end{array}$$

are provable in $\lambda\mathbf{HA}^\omega$.

A partial equivalence relation is an equivalence relation on its domain

$$x \in \mathbf{Dom}_{\mathscr{R}} \quad \equiv \quad x\mathscr{R}x$$

Moreover, using symmetry and transitivity, one can show that

$$x\mathscr{R}y \Rightarrow x \in \mathbf{Dom}_{\mathscr{R}} \wedge y \in \mathbf{Dom}_{\mathscr{R}}$$

Therefore, a partial equivalence relation on σ is exactly an equivalence relation on a collection of individuals of sort σ (i.e. a formula with one free variable of sort σ).

Let $\{=^{\mathrm{ext}}_\sigma\}_\sigma$ and $\{=^{\mathrm{pm}}_\sigma\}_\sigma$ be two families of binary relations indexed by the sorts of System T and defined as follows:

$$\begin{array}{rcl} x^{\mathbf{N}} =^{\mathrm{ext}}_{\mathbf{N}} y^{\mathbf{N}} & \equiv & x = y \\ f^{\sigma\to\tau} =^{\mathrm{ext}}_{\sigma\to\tau} g^{\sigma\to\tau} & \equiv & \forall x\, f\,x =^{\mathrm{ext}}_\tau g\,x \end{array} \qquad \begin{array}{rcl} x^{\mathbf{N}} =^{\mathrm{pm}}_{\mathbf{N}} y^{\mathbf{N}} & \equiv & x = y \\ f^{\sigma\to\tau} =^{\mathrm{pm}}_{\sigma\to\tau} g^{\sigma\to\tau} & \equiv & \forall x \forall y\, x =^{\mathrm{pm}}_\sigma y \Rightarrow f\,x =^{\mathrm{pm}}_\tau g\,y \end{array}$$

Note that

1. The relation $=^{\text{ext}}$ is obtained from equality by extending it to higher sorts in an extensional fashion (two functions are in $=^{\text{ext}}$ if they are extensionally equal).
2. The relation $=^{\text{pm}}$ is obtained from equality by extending it to higher sorts in a parametric fashion (two functions are in $=^{\text{pm}}$ if they send related entries to related outputs).

With an (external) induction on the sorts of System T, it can be shown that for all sorts σ:

1. $=^{\text{ext}}_{\sigma}$ is an equivalence relation;
2. $=^{\text{pm}}_{\sigma}$ is a partial equivalence relation.

We exhibit proof terms that are used on forthcoming translations:

$$\begin{array}{lcl} \vdash \ \mathrm{sym}^{\mathrm{pm}}_{\sigma} & : & \mathbf{Sym}_{=^{\mathrm{pm}}_{\sigma}} \\ \vdash \ \mathrm{trans}^{\mathrm{pm}}_{\sigma} & : & \mathbf{Trans}_{=^{\mathrm{pm}}_{\sigma}} \\ \vdash \ \mathrm{refl}^{\mathrm{pm}}_{\sigma} & : & \forall x^{\sigma} \forall y^{\sigma}\, x =^{\mathrm{pm}}_{\sigma} y \Rightarrow (x =^{\mathrm{pm}}_{\sigma} x \wedge y =^{\mathrm{pm}}_{\sigma} y) \end{array}$$

They are defined by induction on the sort of System T:

$$\begin{array}{rcl} \mathrm{sym}^{\mathrm{pm}}_{\mathbf{N}} & \equiv & \lambda x \lambda y. \lambda \xi. \mathrm{peel}(\xi, \hat{z}.(z = x), \mathrm{refl}\, x) \\ \mathrm{sym}^{\mathrm{pm}}_{\sigma \to \tau} & \equiv & \lambda f \lambda g. \lambda \xi. \lambda x \lambda y. \lambda \eta. \mathrm{sym}^{\mathrm{pm}}_{\tau}(f\,y)(g\,x)(\xi\, y\, x\, (\mathrm{sym}^{\mathrm{pm}}_{\sigma}\, x\, y\, \eta)) \\ \mathrm{trans}^{\mathrm{pm}}_{\mathbf{N}} & \equiv & \lambda x \lambda y \lambda z. \lambda \xi \lambda \eta. \mathrm{peel}(\eta, \hat{w}.x = w, \xi) \\ \mathrm{trans}^{\mathrm{pm}}_{\sigma \to \tau} & \equiv & \lambda f \lambda g \lambda h. \lambda \xi \lambda \eta. \lambda x \lambda y. \lambda \chi. \mathrm{trans}_{\tau}(f\,x)(g\,y)(h\,y)(\xi\, x\, y\, \chi)(\eta\, y\, y (\mathrm{trans}_{\sigma}\, y\, x\, y (\mathrm{sym}^{\mathrm{pm}}_{\sigma} x\, y\, \chi) \chi)) \\ \mathrm{refl}^{\mathrm{pm}}_{\sigma} & \equiv & \lambda x^{\sigma} \lambda y^{\sigma} \lambda \xi. (\mathrm{trans}^{\mathrm{pm}}_{\sigma}\, x\, y\, x\, \xi (\mathrm{sym}^{\mathrm{pm}}\, x\, y\, \xi), \mathrm{trans}^{\mathrm{pm}}_{\sigma}\, y\, x\, y\, (\mathrm{sym}^{\mathrm{pm}}\, x\, y\, \xi) \xi) \end{array}$$

One cannot prove inside $\lambda\mathbf{HA}^{\omega}$ that $=^{\text{pm}}_{\sigma}$ is reflexive for all sorts σ, as it can be seen by working in **HRO**. Let quote $\in \mathbf{HRO}_{(\mathbf{N}\to\mathbf{N})\to\mathbf{N}}$ be an index for the identity function[1] and $p, q \in \mathbf{HRO}_{\mathbf{N}\to\mathbf{N}}$ two distinct indexes for the same total unary function. Note that

$$\begin{array}{lcl} \mathbf{HRO} & \vDash & p =^{\mathrm{pm}}_{\mathbf{N}\to\mathbf{N}} q \\ \mathbf{HRO} & \vDash & \{\mathrm{quote}\}(p) \neq^{\mathrm{pm}}_{\mathbf{N}} \{\mathrm{quote}\}(q) \end{array}$$

Consequently

$$\mathbf{HRO} \ \vDash \ \mathrm{quote} \neq^{\mathrm{pm}}_{(\mathbf{N}\to\mathbf{N})\to\mathbf{N}} \mathrm{quote}$$

and $=^{\text{pm}}_{(\mathbf{N}\to\mathbf{N})\to\mathbf{N}}$ is not reflexive in **HRO**.

Because $=^{\text{ext}}_{(\mathbf{N}\to\mathbf{N})\to\mathbf{N}}$ is reflexive, the previous result shows that in **HRO**, $=^{\text{ext}}_{(\mathbf{N}\to\mathbf{N})\to\mathbf{N}}$ is not included in $=^{\text{pm}}_{(\mathbf{N}\to\mathbf{N})\to\mathbf{N}}$. Therefore, one cannot prove in $\lambda\mathbf{HA}^{\omega}$

$$\forall x \forall y\; x =^{\mathrm{ext}}_{(\mathbf{N}\to\mathbf{N})\to\mathbf{N}} y \Rightarrow x =^{\mathrm{pm}}_{(\mathbf{N}\to\mathbf{N})\to\mathbf{N}} y$$

Going one step higher in the hierarchy of sorts, one can show that

$$\forall x \forall y\; x =^{\mathrm{pm}}_{((\mathbf{N}\to\mathbf{N})\to\mathbf{N})\to\mathbf{N}} y \Rightarrow x =^{\mathrm{ext}}_{((\mathbf{N}\to\mathbf{N})\to\mathbf{N})\to\mathbf{N}} y$$

is not provable in $\lambda\mathbf{HA}^{\omega}$. Indeed, consider a variant $\mathbf{HRO^o}$ of **HRO** where natural numbers denote recursive functions that can access an oracle deciding if its entry is an index of the identity function (i.e.

[1] quote is a functional that takes a function as argument and returns its source code

for instance it returns 1 if it is the case and 0 otherwise). Let $n \in \mathbf{HRO^o}_{((\mathbf{N}\to\mathbf{N})\to\mathbf{N})\to\mathbf{N}}$ an index for this oracle and $m \in \mathbf{HRO^o}_{((\mathbf{N}\to\mathbf{N})\to\mathbf{N})\to\mathbf{N}}$ an index of the constant function $x \mapsto 0$. It turns out that

$$\begin{array}{lcl} \mathbf{HRO^o} & \models & n =^{\mathrm{pm}}_{((\mathbf{N}\to\mathbf{N})\to\mathbf{N})\to\mathbf{N}} m \\ \mathbf{HRO^o} & \models & n \neq^{\mathrm{ext}}_{((\mathbf{N}\to\mathbf{N})\to\mathbf{N})\to\mathbf{N}} m \end{array}$$

because the indexes of the identity function are not in the domain of $=^{\mathrm{pm}}_{(\mathbf{N}\to\mathbf{N})\to\mathbf{N}}$.

Finally, in the set-theoretic model $\mathbf{M}$ of $\mathbf{HA}^\omega$ (and in fact in any extensional model), one can show

$$\mathbf{M} \models \forall x \forall y\, x =^{\mathrm{ext}}_\sigma y \Leftrightarrow x =^{\mathrm{pm}}_\sigma y$$

for all σ (by induction on the sorts of System T).

Therefore, in $\lambda\mathbf{HA}^\omega$, one cannot prove that the relations $=^{\mathrm{pm}}$ and $=^{\mathrm{ext}}$ are different. We wrap up all these results in the following theorem.

Theorem 1. *In $\lambda\mathbf{HA}^\omega$ one cannot prove that*

1. $=^{\mathrm{pm}}_{(\mathbf{N}\to\mathbf{N})\to\mathbf{N}}$ *is reflexive;*
2. $=^{\mathrm{ext}}_{(\mathbf{N}\to\mathbf{N})\to\mathbf{N}} \subseteq =^{\mathrm{pm}}_{(\mathbf{N}\to\mathbf{N})\to\mathbf{N}}$*;*
3. $=^{\mathrm{pm}}_{((\mathbf{N}\to\mathbf{N})\to\mathbf{N})\to\mathbf{N}} \subseteq =^{\mathrm{ext}}_{((\mathbf{N}\to\mathbf{N})\to\mathbf{N})\to\mathbf{N}}$*;*
4. $=^{\mathrm{pm}}_\sigma \subsetneq =^{\mathrm{ext}}_\sigma$ *and* $=^{\mathrm{ext}}_\sigma \subsetneq =^{\mathrm{pm}}_\sigma$ *(for any sort σ);*

where the symbols $\subseteq, \subsetneq$ and the property of being reflexive are defined inside $\lambda\mathbf{HA}^\omega$ in the usual way.

3.2 A first translation: from System T into $\lambda\mathbf{HA}^\omega$

Although one cannot prove inside $\lambda\mathbf{HA}^\omega$ that $=^{\mathrm{pm}}$ is reflexive, it can be shown that all closed terms of System T are in its domain. With this goal in mind, we design a translation from System T into $\lambda\mathbf{HA}^\omega$:

$$(\Delta \vdash_{\mathrm{T}} t : \sigma)^{\mathrm{pm}} \quad\rightsquigarrow\quad \Delta^1, \Delta^2; \Delta^{\mathrm{pm}} \vdash t^{\mathrm{pm}} : t^1 =^{\mathrm{pm}}_\sigma t^2$$

1. Declarations of variables in signatures are duplicated. Fixing $i = 1, 2$:

$$\begin{array}{rcl} \emptyset^i & \equiv & \emptyset \\ (\Delta, x^\sigma)^i & \equiv & \Delta^i, (x^i)^\sigma \end{array}$$

   where x^i are fresh distinct variables.

2. Terms of System T are duplicated. Fixing $i = 1, 2$:

$$t^i \equiv t[\theta^i_t]$$

   will denote the term obtained by substituting all free variables x of t by x^i (i.e. θ^i_t is the substitution defined on the free variables of t that associates to a variable x the variable x^i).

3. Signatures of System T are translated into contexts of $\lambda\mathbf{HA}^\omega$:

$$\begin{array}{rcl} \emptyset^{\mathrm{pm}} & \equiv & \emptyset \\ (\Delta, x^\sigma)^{\mathrm{pm}} & \equiv & \Delta^{\mathrm{pm}}, x^{\mathrm{pm}} : x^1 =^{\mathrm{pm}}_\sigma x^2 \end{array}$$

4. Terms of System T are translated into proof terms of $\lambda\mathbf{HA}^\omega$:

$$
\begin{array}{rcl}
(x)^{\mathrm{pm}} & \equiv & x^{\mathrm{pm}} \\
(\lambda x.t)^{\mathrm{pm}} & \equiv & \lambda x^1 \lambda x^2 . \lambda x^{\mathrm{pm}} . t^{\mathrm{pm}} \\
(t\,u)^{\mathrm{pm}} & \equiv & t^{\mathrm{pm}}\, u^1\, u^2\, u^{\mathrm{pm}} \\
0^{\mathrm{pm}} & \equiv & \mathrm{refl}\, 0 \\
(\mathrm{S}\, t)^{\mathrm{pm}} & \equiv & \mathrm{peel}\big(t^{\mathrm{pm}}, \hat{x}.(\mathrm{S}\, t^1 = \mathrm{S}\, x), \mathrm{refl}\,(\mathrm{S}\, t^1)\big) \\
(\mathrm{Rec}\, t\, u\, v)^{\mathrm{pm}} & \equiv & \mathrm{Ind}\big(\hat{x}.(\forall y^{\mathbf{N}}\, x = y \Rightarrow \mathrm{Rec}^\sigma\, t^1\, u^1\, x =^{\mathrm{pm}}_\sigma \mathrm{Rec}^\sigma\, t^2\, u^2\, y), \\
& & \quad \lambda y. \lambda \xi . \mathrm{peel}(\xi, \hat{z}.(t^1 =^{\mathrm{pm}}_\sigma \mathrm{Rec}^\sigma\, t^2\, u^2\, z), t^{\mathrm{pm}}), \\
& & \quad \lambda x. \lambda \eta . \lambda y. \lambda \xi . \mathrm{peel}(\xi, \hat{z}.(u^1 (\mathrm{Rec}\, t^1\, u^1\, x) x =^{\mathrm{pm}}_\sigma (\mathrm{Rec}\, t^2\, u^2\, z)), \\
& & \qquad\qquad u^{\mathrm{pm}} (\mathrm{Rec}\, t^1\, u^1\, x)(\mathrm{Rec}\, t^2\, u^2\, x)(\eta\, x\, (\mathrm{refl}\, x))\, x\, x\, (\mathrm{refl}\, x)), \\
& & \quad v^1) v^2\, v^{\mathrm{pm}}
\end{array}
$$

The translation works as follows:

1. An abstraction in System T is interpreted as 3 abstractions in $\lambda\mathbf{HA}^\omega$: 2 abstractions of first-order variables and one of proof variable. Indeed, $(\lambda x^\sigma.t)^{\mathrm{pm}}$ is of type $\forall x^1 \forall x^2\, x^1 =^{\mathrm{pm}}_\sigma x^2 \Rightarrow t^1 =^{\mathrm{pm}} t^2$ (assuming $\lambda x^\sigma.t : \sigma \to \tau$).
2. Symmetrically, an application in System T is interpreted as 3 applications.
3. Because the relation $=^{\mathrm{pm}}_{\mathbf{N}}$ is merely the equality, 0 and $\mathrm{S}\,t$ are interpreted as equality proofs.
4. Finally, the recursor is interpreted using an induction. During the induction, the synchronization between the two copies of the term v (of sort $\mathbf{N}$) is lost. Therefore, we need an extra generalization in the hypothesis to retrieve that these terms are equal.

Theorem 2. *If $\Delta \vdash_{\mathcal{T}} t : \sigma$ then $\Delta^1, \Delta^2; \Delta^{\mathrm{pm}} \vdash t^{\mathrm{pm}} : t^1 =^{\mathrm{pm}}_\sigma t^2$. In particular, $\vdash t^{\mathrm{pm}} : t =^{\mathrm{pm}}_\sigma t$ for all closed terms of sort σ.*

Proof. By induction on the derivations of System T. □

We deduce from the previous theorem that each closed term of System T is in the domain of $=^{\mathrm{pm}}$. The following terms are used later:

$$\Delta^1, \Delta^2; \Delta^{\mathrm{pm}} \quad \vdash \quad \mathrm{Elim}^i_{\hat{z}.t} : \forall z^1 \forall z^2\, z^1 =^{\mathrm{pm}}_\sigma z^2 \Rightarrow t^i[z^i = z^1] =^{\mathrm{pm}}_\tau t^i[z^i = z^2]$$

for $i = 1, 2$. Note that these proof terms are indexed by a term t and a variable z. These terms are constructed using the previous translation, as follows:

$$
\begin{array}{rcl}
\mathrm{Elim}^1_{\hat{z}.t} & \equiv & \lambda z^1 \lambda z^2 . \lambda z^{\mathrm{pm}} . \mathrm{trans}^{\mathrm{pm}}\, t^1\, t^2 t^1 [z^1 := z^2] t^{\mathrm{pm}} \\
& & \qquad\qquad (\mathrm{sym}^{\mathrm{pm}} t^1 [z^1 := z^2] t^2 t^{\mathrm{pm}} [z^1 := z^2][z^{\mathrm{pm}} := (\mathrm{refl}^{\mathrm{pm}} z^1 z^2 z^{\mathrm{pm}}).2]) \\
\mathrm{Elim}^2_{\hat{z}.t} & \equiv & \lambda z^1 \lambda z^2 . \lambda z^{\mathrm{pm}} . \mathrm{trans}^{\mathrm{pm}}\, t^2 [z^2 := z^1]\, t^1 t^2 \\
& & \qquad\qquad (\mathrm{sym}^{\mathrm{pm}} t^1 t^2 [z^2 := z^1] t^{\mathrm{pm}} [z^2 := z^1][z^{\mathrm{pm}} := (\mathrm{refl}^{\mathrm{pm}}\, z^1 z^2 z^{\mathrm{pm}}).1]) t^{\mathrm{pm}})
\end{array}
$$

They are well typed as soon as $\mathbf{FV}(t) \subseteq \Delta, z$.

4 Interpreting extensional equality: from λE-HA$^\omega$ into λHA$^\omega$

4.1 A preliminary step: a translation from λHA$^\omega$ into λHA$^\omega$

Although all closed terms of System T are in the domain of $=^{\mathrm{pm}}$, one cannot prove $\forall x^\sigma\, x =^{\mathrm{pm}}_\sigma x$ in $\lambda\mathbf{HA}^\omega$ (for $\sigma = (\mathbf{N}\to\mathbf{N})\to\mathbf{N}$ for instance). Nevertheless, building on the intuition of restricting quantifications and on the work of the previous section, we can design a translation

$$(\Delta;\Gamma \vdash M:\Phi)^{\mathrm{pm}} \quad \rightsquigarrow \quad \Delta^1,\Delta^2;\Delta^{\mathrm{pm}},\Gamma^{\mathrm{pm}} \vdash M^{\mathrm{pm}}:\Phi^{\mathrm{pm}}$$

from $\lambda\mathbf{HA}^\omega$ into itself that will serve as a basis to interpret extensional equality.

1. This translation extends the one defined formerly. In particular $\Delta^i,\Delta^{\mathrm{pm}},t^i$ and t^{pm} are already defined.
2. Formulas of $\lambda\mathbf{HA}^\omega$ are translated into formulas of $\lambda\mathbf{HA}^\omega$:

$$\begin{array}{rcl}
(t=u)^{\mathrm{pm}} & \equiv & t^1 =^{\mathrm{pm}}_{\mathbf{N}} u^2\\
\bot^{\mathrm{pm}} & \equiv & \bot\\
(\Phi\Rightarrow\Psi)^{\mathrm{pm}} & \equiv & \Phi^{\mathrm{pm}}\Rightarrow\Psi^{\mathrm{pm}}\\
(\Phi\wedge\Psi)^{\mathrm{pm}} & \equiv & \Phi^{\mathrm{pm}}\wedge\Psi^{\mathrm{pm}}\\
(\forall x^\sigma\Phi)^{\mathrm{pm}} & \equiv & \forall x^1\forall x^2\, x^1 =^{\mathrm{pm}}_\sigma x^2 \Rightarrow \Phi^{\mathrm{pm}}\\
(\exists x^\sigma\Phi)^{\mathrm{pm}} & \equiv & \exists x^1\exists x^2\, x^1 =^{\mathrm{pm}}_\sigma x^2 \wedge \Phi^{\mathrm{pm}}
\end{array}$$

3. Contexts of $\lambda\mathbf{HA}^\omega$ are translated into contexts of $\lambda\mathbf{HA}^\omega$:

$$\begin{array}{rcl}
\emptyset^{\mathrm{pm}} & \equiv & \emptyset\\
(\Gamma,\xi:\Phi)^{\mathrm{pm}} & \equiv & \Gamma^{\mathrm{pm}},\xi:\Phi^{\mathrm{pm}}
\end{array}$$

4. Proof terms of $\lambda\mathbf{HA}^\omega$ are translated into proof terms of $\lambda\mathbf{HA}^\omega$:

$$\begin{array}{rcl}
(\xi)^{\mathrm{pm}} & \equiv & \xi\\
(\lambda\xi.M)^{\mathrm{pm}} & \equiv & \lambda\xi.M^{\mathrm{pm}}\\
(M\,N)^{\mathrm{pm}} & \equiv & M^{\mathrm{pm}}N^{\mathrm{pm}}\\
(M,N)^{\mathrm{pm}} & \equiv & (M^{\mathrm{pm}},N^{\mathrm{pm}})\\
(M.i)^{\mathrm{pm}} & \equiv & M^{\mathrm{pm}}.i\\
(\lambda x.M)^{\mathrm{pm}} & \equiv & \lambda x^1\lambda x^2\lambda x^{\mathrm{pm}}.M^{\mathrm{pm}}\\
(M\,t)^{\mathrm{pm}} & \equiv & M^{\mathrm{pm}}t^1t^2t^{\mathrm{pm}}\\
([t,M])^{\mathrm{pm}} & \equiv & [t^1,[t^2,(t^{\mathrm{pm}},M^{\mathrm{pm}})]]\\
(\mathrm{let}\,[x,\xi]:=M\,\mathrm{in}\,N)^{\mathrm{pm}} & \equiv & \mathrm{let}\,[x^1,\eta]:=M^{\mathrm{pm}}\,\mathrm{in}\,\mathrm{let}\,[x^2,\chi]:=\eta\,\mathrm{in}\,N^{\mathrm{pm}}[x^{\mathrm{pm}}=\chi.1][\xi^{\mathrm{pm}}:=\chi.2]\\
(\mathrm{efq}(M,\Phi))^{\mathrm{pm}} & \equiv & \mathrm{efq}(M^{\mathrm{pm}},\Phi^{\mathrm{pm}})\\
(\mathrm{refl}\,t)^{\mathrm{pm}} & \equiv & t^{\mathrm{pm}}\\
\left(\mathrm{peel}^{t,u}(M,\hat{x}.\Phi,N)\right)^{\mathrm{pm}} & \equiv & \mathrm{peel}\left(M_2^{\mathrm{pm}},\hat{x^2}.\Phi^{\mathrm{pm}}[x^1:=u^1],\mathrm{peel}(M_1^{\mathrm{pm}},\hat{x^1}.\Phi^{\mathrm{pm}}[x^2:=t^2],N^{\mathrm{pm}})\right)\\
\left(\mathrm{Ind}(\hat{x}.\Phi,M,N,t)\right)^{\mathrm{pm}} & \equiv & \mathrm{Ind}(\hat{x}.\forall y\, x=y\Rightarrow\Phi^{\mathrm{pm}}[x^1:=x][x^2:=y],\\
& & \quad \lambda y\lambda\xi.\mathrm{peel}(\xi,\hat{z}.\Phi^{\mathrm{pm}}[x^1:=0][x2:=z],M^{\mathrm{pm}}),\\
& & \quad \lambda x\lambda\eta\lambda y\xi.\mathrm{peel}(\xi,\hat{z}.\Phi^{\mathrm{pm}}[x^1:=\mathrm{S}\,x][x^2:=z],N^{\mathrm{pm}}x\,x\,(\mathrm{refl}\,x)(\eta\,x\,(\mathrm{refl}\,x)),\\
& & \quad t^1)t^2t^{\mathrm{pm}}
\end{array}$$

where in the translation of $\mathrm{peel}(M,\hat{x}.\Phi,N)$, M_i^{pm} denotes a proof of $t^i=u^i$:

$$\begin{array}{rcll}
M_1^{\mathrm{pm}} & \equiv & \mathrm{trans}^{\mathrm{pm}}_{\mathbf{N}}t^1\,u^2\,u^1\,M^{\mathrm{pm}}(\mathrm{sym}^{\mathrm{pm}}_{\mathbf{N}}u^1\,u^2u^{\mathrm{pm}}) & : \; t^1=u^1\\
M_2^{\mathrm{pm}} & \equiv & \mathrm{trans}^{\mathrm{pm}}_{\mathbf{N}}t^2\,t^1\,u^2(\mathrm{sym}^{\mathrm{pm}}_{\mathbf{N}}t^1\,t^2t^{\mathrm{pm}})M^{\mathrm{pm}} & : \; t^2=u^2
\end{array}$$

$$\frac{(\Delta;\Gamma)\ \mathbf{wf} \quad \Delta \vdash_{\mathrm{T}} t : \sigma}{\Delta;\Gamma \vdash_{\mathrm{e}} \mathrm{refl}_\sigma t : t =_\sigma t} \qquad \frac{\Delta;\Gamma \vdash_{\mathrm{e}} M : t =_\sigma u \quad \Delta;\Gamma \vdash_{\mathrm{e}} N : \Phi[x^\sigma := t]}{\Delta;\Gamma \vdash_{\mathrm{e}} \mathrm{peel}_\sigma^{t,u}(M, \hat{x}.\Phi, N) : \Phi[x^\sigma := u]}$$

$$\frac{\Delta;\Gamma \vdash_{\mathrm{e}} M : \forall x^\sigma f\,x =_\tau g\,x}{\Delta;\Gamma \vdash_{\mathrm{e}} \mathrm{ext}_{\sigma,\tau}(M) : f =_{\sigma\to\tau} g} \qquad \frac{\Delta;\Gamma \vdash_{\mathrm{e}} M : f =_{\sigma\to\tau} g \quad \Delta;\Gamma \vdash_{\mathrm{e}} N : t =_\sigma u}{\Delta;\Gamma \vdash_{\mathrm{e}} \mathrm{app}_{\sigma,\tau}(M,t,u,N) : f\,t =_\tau g\,u}$$

Figure 3: Additional typing rules for extensional equality

Here, the translation of peel is ad hoc: it is merely done by using peel on two distinct equalities. However, it will not be the case in the last translation, where we will need an external recursion on the formulas of the source system to interpret it.

The translation of induction follows the same principle as the translation of the recursor done in Section 3.2: because the synchronization between the copies of t is lost, we need to generalize the inductive hypothesis.

Theorem 3. *If* $\Delta;\Gamma \vdash M : \Phi$ *then* $\Delta^1,\Delta^2;\Delta^{\mathrm{pm}},\Gamma^{\mathrm{pm}} \vdash M^{\mathrm{pm}} : \Phi^{\mathrm{pm}}$.

The proof of this theorem is by induction on the derivation of $\lambda\mathbf{HA}^\omega$ and it uses three lemmas:

Lemma 1. *If* $t \cong u$ *then* $t^i \cong u^i$ *for* $i = 1,2$ *and* t,u *terms of System T.*

Lemma 2. *If* $t(x^\sigma)$ *and* u *are terms with* u *of sort* σ*, then* $\big(t[x := u]\big)^i \equiv t^i[x^i := u^i]$ *for* $i = 1,2$.

Lemma 3. *If* $\Phi(x^\sigma)$ *is a formula and* t *is a term of sort* σ*, then* $\big(\Phi[x := t]\big)^{\mathrm{pm}} \equiv \Phi^{\mathrm{pm}}[x^1 := t^1][x^2 := t^2]$.

4.2 Extending equality through parametricity: a translation from $\lambda\mathbf{E}$-$\mathbf{HA}^\omega$ into $\lambda\mathbf{HA}^\omega$

Our next goal is to extend the previous translation to give an interpretation of extensional equality inside $\lambda\mathbf{HA}^\omega$.

Consider an extension $\lambda\mathbf{E}$-$\mathbf{HA}^\omega$ of $\lambda\mathbf{HA}^\omega$ obtained by extending equality in an extensional way to all higher sorts, i.e. by adding

1. atomic formulas $t =_\sigma u$ for all sorts σ;
2. proof terms $(\mathrm{refl}_\sigma t)$, $\mathrm{peel}_\sigma^{t,u}(M, \hat{x}.\Phi, N)$, $\mathrm{ext}_{\sigma,\tau}(M)$ and $\mathrm{app}_{\sigma,\tau}(M,t,u,N)$ for all sorts σ and τ;
3. typing rules for the added proof terms presented in Figure 3 at page 61.

The symbol $\vdash_{\mathrm{e}}$ will be used to denote sequents (and provability) in $\lambda\mathbf{E}$-$\mathbf{HA}^\omega$. The translation $(_)^{\mathrm{pm}}$ can be extended to a translation from $\lambda\mathbf{E}$-$\mathbf{HA}^\omega$ into $\lambda\mathbf{HA}^\omega$. Indeed, one interprets

$$(t =_\sigma u)^{\mathrm{pm}} \quad \equiv \quad t^1 =_\sigma^{\mathrm{pm}} u^2$$

It is then possible to extend the translation with

$$(\mathrm{refl}_\sigma t)^{\mathrm{pm}} \quad \equiv \quad t^{\mathrm{pm}}$$

and still preserving adequacy. The case of $\big(\mathrm{peel}_\sigma^{t,u}(M, \hat{x}.\Phi, N)\big)^{\mathrm{pm}}$ is more involved and it is treated as follows. We first construct a family of terms $\mathrm{Elim}_{\hat{x}.\Phi}$ satisfying that if $\mathbf{FV}(\Phi) \subseteq \Delta, x^{\mathrm{pm}}$ then

$$\Delta^1,\Delta^2;\Delta^{\mathrm{pm}} \quad \vdash \quad \mathrm{Elim}_{\hat{x}.\Phi} : \forall x^1 \forall x^2 \forall y^1 \forall y^2\, x^1 =_\sigma^{\mathrm{pm}} y^1 \Rightarrow x^2 =_\sigma^{\mathrm{pm}} y^2 \Rightarrow \Phi^{\mathrm{pm}} \Rightarrow \Phi^{\mathrm{pm}}[x^1 := y^1][x^2 := y^2]$$

This is done by induction on the syntax of formulas:

$$\begin{array}{rcl}
\mathrm{Elim}_{\hat{x}.t=_\sigma u} & \equiv & \lambda x^1\lambda x^2\lambda y^1\lambda y^2\lambda\xi^1\lambda\xi^2\lambda\xi.\mathrm{trans}^{\mathrm{pm}}\, t^1[x^1 := y^1]\, t^1\, u^2[x^2 := y^2] \\
 & & \quad (\mathrm{Elim}^1_{\hat{x}.t}\, y^1\, x^1\, (\mathrm{sym}^{\mathrm{pm}}\, x^1\, y^1\, \xi^1)) \\
 & & \quad (\mathrm{trans}^{\mathrm{pm}}\, t^1\, u^2\, u^2[x^2 := y^2]\xi\, (\mathrm{Elim}^2_{\hat{x}.u}\, x^2\, y^2\, \xi^2)) \\
\mathrm{Elim}_{\hat{x}.\bot} & \equiv & \lambda x^1\lambda x^2\lambda y^1\lambda y^2\lambda\xi^1\lambda\xi^2\lambda\xi.\xi \\
\mathrm{Elim}_{\hat{x}.(\Phi\Rightarrow\Psi)} & \equiv & \lambda x^1\lambda x^2\lambda y^1\lambda y^2\lambda\xi^1\lambda\xi^2\lambda\xi.\lambda\eta.\mathrm{Elim}^+_{\hat{x}.\Psi}(\xi\ (\mathrm{Elim}^-\eta)) \\
\mathrm{Elim}_{\hat{x}.(\Phi\wedge\Psi)} & \equiv & \lambda x^1\lambda x^2\lambda y^1\lambda y^2\lambda\xi^1\lambda\xi^2\lambda\xi.(\mathrm{Elim}^+_{\hat{x}.\Phi}\xi.1, \mathrm{Elim}^+_{\hat{x}.\Psi}\xi.2) \\
\mathrm{Elim}_{\hat{x}.(\forall z\Phi)} & \equiv & \lambda x^1\lambda x^2\lambda y^1\lambda y^2\lambda\xi^1\lambda\xi^2\lambda\xi.\lambda z^1\lambda z^2\lambda z^{\mathrm{pm}}.\mathrm{Elim}_{\hat{x}.\Phi}(\xi\, z^1 z^2 z^{\mathrm{pm}}) \\
\mathrm{Elim}_{\hat{x}.(\exists z\Phi)} & \equiv & \lambda x^1\lambda x^2\lambda y^1\lambda y^2\lambda\xi^1\lambda\xi^2\lambda\xi.\mathrm{let}\,[z^1,\eta] := \xi\,\mathrm{in}\,\mathrm{let}\,[z^2,\chi] := \eta\,\mathrm{in}\,[z^1,[z^2,(\eta.1, \mathrm{Elim}^+_{\hat{x}.\Phi}\eta.2)]]
\end{array}$$

where

$$\begin{array}{rcl}
\mathrm{Elim}^+_{\hat{x}.\Phi} & \equiv & \mathrm{Elim}_{\hat{x}.\Phi}\, x^1\, x^2\, y^1\, y^2\, \xi^1\, \xi^2 \\
\mathrm{Elim}^- & \equiv & \mathrm{Elim}_{\hat{y}.\Phi[x:=y]}\, y^1\, y^2\, x^1\, x^2\, (\mathrm{sym}^{\mathrm{pm}}\, x^1\, y^1\, \xi^1)(\mathrm{sym}^{\mathrm{pm}}\, x^2\, y^2\, \xi^2)
\end{array}$$

The proof that $\mathrm{Elim}_{\hat{x}.\Phi}$ satisfies the given property is by induction on the syntax, where the hypothesis is used to treat the case of equality.

We can now define

$$\begin{array}{rcl}
\big(\mathrm{peel}^{t,u}_\sigma(M,\hat{x}.\Phi,N)\big)^{\mathrm{pm}} & \equiv & \mathrm{Elim}_{\hat{x}.\Phi} t^1 t^2 u^1 u^2 (\mathrm{trans}^{\mathrm{pm}} t^1 u^2 u^1 M^{\mathrm{pm}} (\mathrm{sym}^{\mathrm{pm}} u^1 u^2 u^{\mathrm{pm}})) \\
 & & \quad (\mathrm{trans}^{\mathrm{pm}} t^2 t^1 u^2 (\mathrm{sym}^{\mathrm{pm}} t^1 t^2 t^{\mathrm{pm}}) M^{\mathrm{pm}}) N^{\mathrm{pm}}
\end{array}$$

Finally, we set

$$\begin{array}{rcl}
\big(\mathrm{ext}_{\sigma\to\tau}(M)\big)^{\mathrm{pm}} & \equiv & \lambda x^1, x^2 \lambda x^{\mathrm{pm}}.M^{\mathrm{pm}}\, x^1\, x^2\, x^{\mathrm{pm}} \\
\big(\mathrm{app}_{\sigma\to\tau}(M,t,u,N)\big)^{\mathrm{pm}} & \equiv & M^{\mathrm{pm}}\, t^1\, u^2\, N^{\mathrm{pm}}
\end{array}$$

Theorem 4. *If* $\Delta;\Gamma\vdash_{\mathrm{e}} M:\Phi$ *then* $\Delta^1,\Delta^2;\Delta^{\mathrm{pm}},\Gamma^{\mathrm{pm}}\vdash M^{\mathrm{pm}}:\Phi^{\mathrm{pm}}$.

Proof. The case of $\big(\mathrm{peel}_\sigma(M,\hat{x}.\Phi,N)\big)^{\mathrm{pm}}$ uses the property of $\mathrm{Elim}_{\hat{x}.\Phi}$ and a generalization of Fact 4, saying that if $\Delta;\Gamma\vdash_{\mathrm{e}} M:\Phi$ then $\mathbf{FV}(\Phi)\subseteq\Delta$. □

Corollary 1. *If* λ**HA**$^\omega$ *is consistent, then so is* λ**E**-**HA**$^\omega$.

Proof. If λ**E**-**HA**$^\omega$ is inconsistent, there exists a proof term M such that $\vdash_e M:\bot$. By the previous translation, one gets a derivation of $\vdash M^{\mathrm{pm}}:\bot$ and concludes that λ**HA**$^\omega$ is inconsistent. □

4.3 Characterizing the image of the translation

In the previous section, we showed that if a closed formula Φ is provable in λ**E**-**HA**$^\omega$ then Φ^{pm} is provable in λ**HA**$^\omega$. The goal of this section is to prove the converse: if Φ^{pm} is provable in λ**HA**$^\omega$ then Φ is provable in λ**E**-**HA**$^\omega$. These properties show that the system λ**E**-**HA**$^\omega$ fully characterizes the image of the translation we designed.

We first show that the relation $=^{\mathrm{pm}}$ collapses to the equality relation in λ**E**-**HA**$^\omega$. For every sort σ, we construct a proof term

$$\vdash_e \mathrm{Collaps}_\sigma : \forall x^\sigma \forall y^\sigma\ x =_\sigma y \Leftrightarrow x =^{\mathrm{pm}}_\sigma y$$

by external induction on the sorts of System T:

$$\begin{array}{rcl}
\mathrm{Collaps}_{\mathbf{N}} & \equiv & \lambda x\lambda y\big(\lambda\xi.\xi, \lambda\xi.\xi\big) \\
\mathrm{Collaps}_{\sigma\to\tau} & \equiv & \lambda f\lambda g(\lambda\xi\lambda x\lambda y\lambda\eta.\mathrm{Collaps}_\tau.1\,(f\,x)\,(g\,y)\,\mathrm{app}_{\sigma,\tau}(\xi,x,y,\mathrm{Collaps}_\sigma.2\,x\,y\,\eta), \\
 & & \quad \lambda\xi.\mathrm{ext}_{\sigma,\tau}(\lambda z.\mathrm{Collaps}_\tau.2\,(f\,z)\,(g\,z)(\xi\,z\,z(\mathrm{Collaps}_\sigma.1\,z\,z,(\mathrm{refl}\,z)))))
\end{array}$$

Proposition 1. *For every sort σ, the binary relations $=^{\mathrm{pm}}_\sigma$ and $=^{\mathrm{ext}}_\sigma$ collapse to $=_\sigma$ in λE-HA$^\omega$.*

Proof. We proved above that $=^{\mathrm{pm}}_\sigma$ collapses to $=_\sigma$ and it can be proved in a similar fashion that $=^{\mathrm{ext}}_\sigma$ also collapses to $=_\sigma$ in λ**E-HA**$^\omega$. □

Using the Proposition 1, we can now show that the image of the last translation is fully characterized by the type system λ**E-HA**$^\omega$.

We exhibit a family of proof terms Equiv^i_Φ for $i = 1,2$ satisfying, for any formula Φ and any signatures Δ containing the free variables of Φ, that

$$\begin{array}{lcl} \Delta^1,\Delta^2;\Delta^{\mathrm{pm}} & \vdash_e & \mathrm{Equiv}^1_\Phi : \Phi^1 \Rightarrow \Phi^{\mathrm{pm}} \\ \Delta^1,\Delta^2;\Delta^{\mathrm{pm}} & \vdash_e & \mathrm{Equiv}^2_\Phi : \Phi^{\mathrm{pm}} \Rightarrow \Phi^1 \end{array}$$

Such proof terms are defined as follows:

$$\begin{array}{rcl}
\mathrm{Equiv}^1_{t=_\sigma u} & \equiv & \lambda\xi.\mathrm{trans}_\sigma\, t^1\, u^1\, u^2 (\mathrm{Collaps}_\sigma.1\, t^1\, u^1\, \xi)\, u^{\mathrm{pm}} \\
\mathrm{Equiv}^2_{t=_\sigma u} & \equiv & \lambda\xi.\mathrm{Collaps}_\sigma.2\, t^1\, u^1\, (\mathrm{trans}_\sigma\, t^1\, u^2\, u^1\, \xi\, (\mathrm{sym}^{\mathrm{pm}}\, u^1\, u^2\, u^{\mathrm{pm}})) \\
\\
\mathrm{Equiv}^1_{\Phi\Rightarrow\Psi} & \equiv & \lambda\xi\lambda\eta.\mathrm{Equiv}^1_\Psi(\xi\, (\mathrm{Equiv}^2_\Phi\, \eta)) \\
\mathrm{Equiv}^2_{\Phi\Rightarrow\Psi} & \equiv & \lambda\xi\lambda\eta.\mathrm{Equiv}^2_\Psi(\xi\, (\mathrm{Equiv}^1_\Phi\, \eta)) \\
\\
\mathrm{Equiv}^1_{\Phi\wedge\Psi} & \equiv & \lambda\xi.(\mathrm{Equiv}^1_\Phi\xi.1, \mathrm{Equiv}^1_\Psi\xi.2) \\
\mathrm{Equiv}^2_{\Phi\wedge\Psi} & \equiv & \lambda\xi.(\mathrm{Equiv}^2_\Phi\xi.1, \mathrm{Equiv}^2_\Psi\xi.2) \\
\\
\mathrm{Equiv}^1_{\forall x^\sigma\Phi} & \equiv & \lambda\xi\lambda x^1\lambda x^2.\lambda x^{\mathrm{pm}}.\mathrm{Equiv}^1_\Phi\, (\xi\, x^1) \\
\mathrm{Equiv}^2_{\forall x^\sigma\Phi} & \equiv & \lambda\xi\lambda x^1.\mathrm{Equiv}^2_\Phi[x_2 := x_1][x^{\mathrm{pm}} := (\mathrm{Collaps}_\sigma.1\, x^1\, x^1\, (\mathrm{refl}_\sigma\, x^1))] \\
 & & \quad (\xi\, x^1\, x^1\, (\mathrm{Collaps}_\sigma.1\, x^1\, x^1\, (\mathrm{refl}_\sigma\, x^1))) \\
\\
\mathrm{Equiv}^1_{\exists x^\sigma\Phi} & \equiv & \lambda\xi.\mathrm{let}\, [x,\eta] := \xi\, \mathrm{in}\, [x, [x, (\mathrm{Collaps}_\sigma.1\, x\, x\, (\mathrm{refl}_\sigma\, x), \mathrm{Equiv}^1_\Phi\xi)]] \\
\mathrm{Equiv}^2_{\exists x^\sigma\Phi} & \equiv & \lambda\xi.\mathrm{let}\, [x^1,\eta] := \xi\, \mathrm{in}\, \mathrm{let}\, [x^2,\chi] := \eta\, \mathrm{in}\, [x^1, \mathrm{Equiv}^2_\Phi\chi.2]
\end{array}$$

We can then conclude that a closed formula Φ is provable in λ**E-HA**$^\omega$ if and only if its translation is provable in λ**E-HA**$^\omega$.

Theorem 5. *For a closed formula Φ of λ**E-HA**$^\omega$*

$$\vdash_e (\mathrm{Equiv}^1_\Phi, \mathrm{Equiv}^2_\Phi) : \Phi \Leftrightarrow \Phi^{\mathrm{pm}}$$

*In particular, if Φ^{pm} is provable in λ**HA**$^\omega$ then Φ is provable in λ**E-HA**$^\omega$.*

4.4 Adding reduction rules: a conjecture

The proof systems used here lack of computational rules, such as

$$\begin{array}{rcl}
(\lambda x.M)\, t & \succ_\beta & M[x := t] \\
(\lambda\xi.M)\, N & \succ_\beta & M[\xi := N] \\
\mathrm{let}\, [x,\xi] := [t,M]\, \mathrm{in}\, N & \succ_\beta & N[x := t][\xi := M] \\
(M_1,M_2).i & \succ_\beta & M.i \\
\mathrm{Ind}(\hat{x}.\Phi, M, N, 0) & \succ_\iota & M \\
\mathrm{Ind}(\hat{x}.\Phi, M, N, S\,t) & \succ_\iota & N\, t\, \mathrm{Ind}(\hat{x}.\Phi, M, N, t) \\
\mathrm{peel}(\mathrm{refl}\, t, \hat{x}.\Phi, N) & \succ_\iota & N
\end{array}$$

and in $\lambda\mathbf{E}$-$\mathbf{HA}^\omega$

$$\mathrm{app}(\mathrm{ext}(M), t, t, \mathrm{refl}\, t) \quad \succ \quad M\, t$$

One could try to figure out if the translation $(_)^{\mathrm{pm}}$ respects reductions, i.e. a property of the shape $M \rightsquigarrow N$ implies $M^{\mathrm{pm}} \simeq_{\beta,\iota} N^{\mathrm{pm}}$. While this is true for β-reductions, it seems that it does not hold for the reduction of $\mathrm{Ind}(\hat{x}.\Phi, M, N, \mathrm{S}\,t)$ if t contains free first-order variables. Indeed, the term $\mathrm{S}\,t$ will be translated as a proof term asserting an equality and if it does not compute into $\mathrm{refl}\,(\mathrm{S}\,t)$, it will block the reduction of the subterm peel(...) inside $(\mathrm{Ind}(\hat{x}.\Phi, M, N, St))^{\mathrm{pm}}$. In the case of a closed term t, we conjecture it will compute as desired. Concretely, we think that the proof will use the meta properties of System T described at the end of Section 2.1: one will use that every closed term of sort $\mathbf{N}$ is β-equivalent to a term of the shape $\mathrm{S}^n\,0$ and that the translation of such a term computes into $\mathrm{refl}\,(\mathrm{S}^n\,0)$, assuming that the translation from System T to $\lambda\mathbf{HA}^\omega$respects reductions.

Conjecture 1. *If M is a proof term of $\lambda\mathbf{E}$-$\mathbf{HA}^\omega$ without free first-order variables and if M does not contain* peel *then $M \rightsquigarrow N$ implies $M^{\mathrm{pm}} \simeq_{\beta,\iota} N^{\mathrm{pm}}$.*

However, the case of the reduction rule of peel seems more difficult to treat, as the translation of peel relies on an external induction over the syntax but also because it uses proofs of symmetry and transitivity of $=^{\mathrm{pm}}$ that do not seem to compute as needed.

5 Related work

The idea to build a syntactic model satisfying extensionality axioms is already present in Gandy's work [4]. Concretely, in higher-order logic, Gandy defined a syntactic model by restricting the elements of discourse to *parametric* ones and proved that, in this model, extensionally equal elements satisfy the same properties. Here, we adapt this construction to the theory $\mathbf{HA}^\omega$ and, using ideas of the Curry-Howard correspondence, we formulate it as a translation of proof systems. Our translation slightly differs from the one of Gandy because we use techniques from parametricity. This choice comes from the idea that parametric translation can serve to extend equality. Nevertheless, because extensionality and parametricity relations collapse to equality in an extensional model (as shown in Proposition 1), it is somehow a matter of design: the translation from $\lambda\mathbf{E}$-$\mathbf{HA}^\omega$ into $\lambda\mathbf{HA}^\omega$ could be designed without the use of parametricity.

Zucker already proved a result of relative consistency between $\mathbf{E}$-$\mathbf{HA}^\omega$ and $\mathbf{N}$-$\mathbf{HA}^\omega$ [12, 10]. He did it in a semantical fashion by transforming models of $\mathbf{N}$-$\mathbf{HA}^\omega$ into models of $\mathbf{E}$-$\mathbf{HA}^\omega$. The method he used is similar to the method of Gandy but, in this context, it suffices to restrict the domain to *parametric* elements and to check that the relation of extensionality is an equivalence relation that is congruent (thus suited to interpret equality). In our work, rather than from $\mathbf{N}$-$\mathbf{HA}^\omega$, we start with $\mathbf{HA}^\omega$: we reconstruct the equality from scratch and show that it respects Leibniz principle. Despite this slight difference, our work can be seen as the syntactical counterpart of the result of Zucker. One advantage is that a syntactical translation comes with an explicit translation of proofs, that we formulate here as a translation of proof terms.

The ideas behind the proof system $\lambda\mathbf{HA}^\omega$ are folklore. For instance, representing the axiom scheme of induction as an inference rule can be seen in many other proof systems, as for instance in Martin-Löf Type Theory [6]. The idea to use the predicate $\mathrm{null}(t)$ to derive Peano's fourth axiom already appears in Miquel's work [7]. The terminology "peel" that we use to denote the eliminator of equality is similar to the one used in some presentations of type theory [8]; however our own motivation to use it is to emphasize that Leibniz principle is recovered by doing an external induction on the formulas or, more graphically, by peeling out the syntax.

6 Conclusion

We designed a translation from $\lambda\mathbf{E}\text{-}\mathbf{HA}^{\omega}$ into $\lambda\mathbf{HA}^{\omega}$ using techniques reminiscent of parametricity and we proved a result of relative consistency: if $\lambda\mathbf{HA}^{\omega}$ is consistent, then so is $\lambda\mathbf{E}\text{-}\mathbf{HA}^{\omega}$. The following diagram shows an intuition of the translation:

$\lambda\mathbf{E}\text{-}\mathbf{HA}^{\omega}$		$\lambda\mathbf{HA}^{\omega}$
t	$\rightsquigarrow$	$t^1 \overset{t^{pm}}{=\!=} t^2$
$M : t = u$	$\rightsquigarrow$	$t^1 \overset{t^{pm}}{=\!=} t^2$, $u^1 \overset{u^{pm}}{=\!=} u^2$, $t^1 \overset{M^{pm}}{=\!=} u^2$

A first-order term t is interpreted as a proof of (parametric) equality between two copies of itself, and an equality proof $M : t = u$ is translated into a proof of (parametric) equality between a copy of t and a copy of u. While the choice to translate M as an equality between t^1 and u^2 is ad hoc (in the sense that it is not imposed by the translation), it is notable that equality proofs are translated into (parametric) equality proofs without the need of higher-order structures (that do not exist in this framework).

References

[1] T. Altenkirch, S. Boulier, A. Kaposi & N. Tabareau (2019): *Setoid Type Theory—A Syntactic Translation*. In: *Mathematics of Program Construction*, Springer International Publishing, pp. 155–196, doi:10.1007/978-3-642-41582-1_10.

[2] J.P. Bernardy, P. Jansson & R. Paterson (2010): *Parametricity and Dependent Types*. *ACM SIGPLAN Notices* 45, pp. 345–356, doi:10.1145/1932681.1863592.

[3] S. Boulier, P.M. Pédrot & N. Tabareau (2017): *The next 700 syntactical models of type theory*. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, Association for Computing Machinery, New York, NY, USA, p. 182–194, doi:10.1145/3018610.3018620.

[4] R. O. Gandy (1956): *On the Axiom of Extensionality–Part I*. *The Journal of Symbolic Logic* 21(1), pp. 36–48, doi:10.1073/pnas.24.12.556.

[5] J.Y. Girard, Y. Lafont & P. Taylor (1989): *Proofs and Types*. *Cambridge Tracts in Theoretical Computer Science* 7, Cambridge University Press.

[6] P. Martin-Löf (1984): *Intuitionistic Type Theory*. Bibliopolis.

[7] A. Miquel (2009): *Relating Classical Realizability and Negative Translation for Existential Witness Extraction*. In: *Typed Lambda Calculi and Applications*, Springer Berlin Heidelberg, pp. 188–202, doi:10.1007/978-3-540-74915-8_25.

[8] B. Nordström, K. Petersson & J.M. Smith (1990): *Programming in Martin-Löf's Type Theory: An Introduction*. Clarendon Press, USA.

[9] J.C. Reynolds (1983): *Types, Abstraction and Parametric Polymorphism*. In: *IFIP Congress*, doi:10.1007/3-540-55511-0_1.

[10] A.S. Troelstra (1973): *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. New York: Springer, doi:10.1007/BFb0066739.

[11] P. Wadler (1989): *Theorems for Free!* In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, Association for Computing Machinery, New York, NY, USA, p. 347–359, doi:10.1145/99370.99404.

[12] J. I. Zucker (1971): *Proof theoretic studies of systems of iterated inductive definitions and subsystems of analysis.* Ph.D. thesis, Standford University.