

The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations

Part 2—A Survey

Amy P. Felty · Alberto Momigliano ·
Brigitte Pientka

March 19, 2014

Abstract Over the past three decades, a variety of meta-reasoning systems which support reasoning about higher-order abstract specifications have been designed and developed. In this paper, we survey and compare four meta-reasoning systems, Twelf, Beluga, Abella and Hybrid, using several benchmarks from the open repository ORBI that describes challenge problems for reasoning with higher-order abstract syntax representations. In particular, we investigate how these systems mechanize and support reasoning using a context of assumptions. This highlights commonalities and differences in these systems and is a first step towards translating between them.

Keywords Logical Frameworks · Higher-Order Abstract Syntax · Proof Assistants · Benchmarks

1 Introduction

Mechanizing formal systems and proofs about them plays an increasingly important role in designing safe, reliable, and trustworthy software systems in general, and in programming languages in particular, see for example LLVM (Zhao et al, 2012), VTS (Appel, 2011), including CompCert (Leroy, 2009) to name a few. A key question in this endeavor is how to represent the given object language (OL) in a given meta-language (i.e. the language of the proof assistant). While a wide range of approaches to encoding OLs exist, they vary substantially in how much generic support they provide for common details and bureaucratic infrastructure.

A. P. Felty
School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Canada, E-mail: afelty@eecs.uottawa.ca

A. Momigliano
Dipartimento di Informatica, Università degli Studi di Milano, Italy, E-mail: momigliano@di.unimi.it

B. Pientka
School of Computer Science, McGill University, Montreal, Canada, E-mail: bpientka@cs.mcgill.ca

Higher-order abstract syntax (HOAS) encodings represent variables in the OL via variables in the meta-language and inherit thereby α -renaming and term-level substitution. Moreover, this encoding technique scales to representing formal systems that use hypothetical and parametrical reasoning by providing generic support for managing hypotheses and the corresponding substitution lemmas. HOAS encodings aim to relieve users from having to build up common infrastructure dealing with variables, assumptions, and substitutions. We consider it the most advanced encoding technique for prototyping formal systems, since it enables us to tackle challenging problems concisely without substantial overhead.

This paper aims to survey the state of the art of several systems supporting HOAS encodings by implementing the ORBI benchmarks (Open challenge problem Repository for systems supporting reasoning with Binders) that we have described in the companion paper (Felty et al, 2014). While we have tried to make the present paper self-contained, the kind reader would greatly benefit from reading both papers hand in hand. These benchmark problems are specifically designed to highlight reasoning using a context of assumptions. This allows us to understand this one key aspect, in which all these systems differ.

In this paper, we concentrate on the logical framework Twelf (Schürmann, 2009), the functional dependently-typed language Beluga (Pientka, 2008; Pientka and Dunfield, 2010), the interactive theorem prover Abella (Gacek, 2008), and the interactive theorem proving environment Hybrid (Momigliano et al, 2008; Felty and Momigliano, 2012), and show how each of these systems implements the ORBI benchmark problems.

Surveying the state of the art in this area serves as a starting point to understand the commonalities and differences between systems. In particular, we concentrate on the following questions:

- How are contexts represented?
- Which structural properties are enforced and how?
- Do structural properties need to be established individually?
- How do we ensure that all elements in a context are unique?
- How is the substitution property guaranteed?
- How do we relate and reason about different contexts?

We see this as a first step towards translating between the different systems. This paper is intended neither to be a tutorial for coding deductive systems with HOAS, for which many excellent ones are readily available, nor as an introduction to the basics of the systems we discuss—in fact, we assume a passing familiarity with them.

We start in Section 2 with a recap of the benchmarks proposed in the companion paper (Felty et al, 2014). Then, in Sections 3 and 4, we discuss the formalizations in Twelf and Beluga, respectively. In Section 5, we present the specification logic used by both Hybrid and Abella, and in Sections 6 and 7, we review the formalization of the benchmarks in these two systems, respectively. Section 8 draws some comparison. Finally, we sum up in Section 9. An electronic appendix ([jar-eappendix.pdf](#)) as well as full details about the challenge problems and their mechanization can be found at <https://github.com/pientka/ORBI/>. All references to the appendix in this paper point to this electronic appendix.

2 Overview of Benchmarks

Encoding an object logic (OL) in a logical framework involves formally representing the OL's syntax, judgments, inference rules, and context structure. In this paper we use the ORBI language proposed in the companion paper (Felty et al, 2014) to present the OLs that constitute our benchmark problems. The ORBI syntax is currently under development, with the goal of providing a uniform representation of OLs that can be fairly directly translated to systems supporting reasoning with HOAS. In its current version, it supports all of the systems considered in this paper. We give some brief examples in Section 2.1, and refer the reader to Appendix A for all other specifications.

In Section 2.2, we list all of the benchmark theorems from Section 3 of the companion paper using the same numbering as in that paper, so that we may refer to them in subsequent sections as we discuss their proofs in the different systems. We give only a brief explanation of these theorems here.

2.1 Object Logic Specifications

We start with some examples taken from the first benchmark in the companion paper, which considers establishing reflexivity, symmetry, and transitivity of algorithmic equality for the untyped lambda-calculus. The **Syntax**, **Judgments**, and **Rules** sections of an ORBI file adopt the concrete syntax of the Logical Framework (LF) (Harper et al, 1993). We recall from Section 4.1 of the companion paper that untyped lambda-terms are introduced with the declaration `tm:type`, followed by the declarations of the constructors `app` with type `tm -> tm -> tm`, and `lam` with type `(tm -> tm) -> tm` in the **Syntax** section. (See Appendix A.1.) The type for `lam` reflects the fact that we represent binders in the object language using binders in the HOAS meta-language.

A predicate representing the algorithmic equality judgment is introduced in the **Judgments** section, followed by object-level inference rules for these judgments in the **Rules** section. For algorithmic equality, we have the predicate `aeq` of type `tm -> tm -> type`, and the following two inference rules, where the ORBI text is a straightforward HOAS encoding of the associated rules.

$$\begin{array}{l}
 \text{ae_a: } \text{aeq } M_1 \ N_1 \ \rightarrow \ \text{aeq } M_2 \ N_2 \\
 \quad \rightarrow \ \text{aeq } (\text{app } M_1 \ M_2) \ (\text{app } N_1 \ N_2) .
 \end{array}
 \qquad
 \frac{\text{aeq } M_1 \ N_1 \quad \text{aeq } M_2 \ N_2}{\text{aeq } (\text{app } M_1 \ M_2) \ (\text{app } N_1 \ N_2)} \text{ae}_a$$

$$\begin{array}{l}
 \text{ae_l: } (\{x:\text{tm}\} \ \text{aeq } x \ x \ \rightarrow \ \text{aeq } (M \ x) \ (N \ x)) \\
 \quad \rightarrow \ \text{aeq } (\text{lam } (\lambda x. M \ x)) \ (\text{lam } (\lambda x. N \ x)) .
 \end{array}
 \qquad
 \frac{\Gamma, x : \text{tm}; \text{aeq } x \ x \vdash \text{aeq } M \ N}{\Gamma \vdash \text{aeq } (\text{lam } x. M) \ (\text{lam } x. N)} \text{ae}_l$$

Section 2 of the companion paper presents a theory of contexts of assumptions that we use in the benchmarks. There, we present context formation rules defining the structure of contexts and discuss structural properties that contexts should satisfy. We recall here that contexts are made up of blocks, the semi-colon is used as a separator within blocks, and the comma is used to separate blocks. This structure can be seen in the hypothesis of the ae_l rule above. In general *context schemas* are used to specify such context structure (see Felty et al, 2014, Section 2.2). In the algorithmic equality benchmark, the following two context schemas are declared in the ORBI **Schemas** section.

```

schema xG: block (x:tm).
schema xAG: block (x:tm; u:aeq x x).

```

Moreover, *context relations* (see Felty et al, 2014, Section 2.4) are defined inductively in ORBI in the **Definitions** section. For example, the following definition relates two contexts satisfying the above two schemas.

```

inductive xAR: {G:xG} {H:xAG} prop =
| xa_nil: xAR nil nil
| xa_cons: xAR G H -> xAR (G, block (x:tm)) (H, block (x:tm; u:aeq x x)).

```

In the next subsection we will write $(xG \Phi_x)$ and $(xAG \Phi_{xa})$ to mean that the contexts Φ_x and Φ_{xa} satisfy the above two schemas, respectively, and $(xAR \Phi_x \Phi_{xa})$ to say that the pair of contexts satisfies the above relation.

Contexts schemas can have *alternatives*. For example, one of our benchmark problems extends the notion of algorithmic equality to the polymorphically-typed lambda-calculus. In addition to **app** and **lam**, this OL includes the constants **tapp** of type $tm \rightarrow tp \rightarrow tm$ and **tlam** of type $(tp \rightarrow tm) \rightarrow tm$ for type application and type abstraction, respectively, where **tp** is the constant introduced to represent object-level types. (See Appendix A.2.) The following two context schemas from that benchmark include an alternative for types, and thus contexts satisfying these schemas may contain blocks of both forms.

```

schema axG: block (a:tp) + block (x:tm).
schema aeqG: block (a:tp; u:atp a a) + block (x:tm; v:aeq x x).

```

In the formal proofs in the rest of this paper, we will refer to the structural properties of contexts in Section 2.3 of the companion paper. We do not repeat them here, but just note that we distinguish between weakening, strengthening, and exchange within a single block and over the context as a whole. The former are called *d-wk*, *d-str*, and *d-exc*, which the latter are called *c-wk*, *c-str*, and *c-exc*.

2.2 A Recap of the Benchmark Theorems

The benchmarks in the companion paper are structured around different shapes and properties of contexts. To help the reader we repeat here the classification, after some initial considerations, but we refer to the companion paper for its motivations. All of these benchmarks include both G versions and R versions of the theorem statements. These two approaches are discussed in detail in the companion paper. Here, we simply note that the G approach uses a common context for all judgments in the statement of a theorem, while the R version may use different contexts and relate them. The reader is referred to the electronic appendix for all definitions of context schemas and context relations.

Inference rules for well-formed terms and types were presented in Section 2.4 of the companion paper. Such inference rules can be inferred from the declarations in the **Syntax** section of an ORBI specification. We do not present them explicitly, but just note here that we name them by prefixing **is_** to the name. In the statements below, we have **is_tm** and **is_tp**. In general, these predicates are used in theorem statements when the proof involves induction over the well-formedness inference rules (e.g., Theorems 7 and 8).

Basic Linear Context Extensions The first benchmark (algorithmic equality for the untyped lambda-calculus) includes Lemmas and Theorems 6–10. See Appendix A.1 and Felty et al (2014, Section 3.1).

Lemma 6 (Context Membership)

$(\mathbf{x}\mathbf{a}\mathbf{R} \Phi_x \Phi_{x_a})$ implies that $\mathbf{x}:\mathbf{tm} \in \Phi_x$ iff $(\mathbf{x}:\mathbf{tm}; \mathbf{a}\mathbf{e}\mathbf{q} \mathbf{x} \mathbf{x}) \in \Phi_{x_a}$.

Theorem 7 (Admissibility of Reflexivity, R Version)

Assume $(\mathbf{x}\mathbf{a}\mathbf{R} \Phi_x \Phi_{x_a})$. If $[\Phi_x \mid - \mathbf{is_tm} \mathbf{M}]$ then $[\Phi_{x_a} \mid - \mathbf{a}\mathbf{e}\mathbf{q} \mathbf{M} \mathbf{M}]$.

Theorem 8 (Admissibility of Reflexivity, G Version)

Assume $(\mathbf{x}\mathbf{a}\mathbf{G} \Phi_{x_a})$. If $[\Phi_{x_a} \mid - \mathbf{is_tm} \mathbf{M}]$ then $[\Phi_{x_a} \mid - \mathbf{a}\mathbf{e}\mathbf{q} \mathbf{M} \mathbf{M}]$.

Lemma 9 (Context Inversion)

Assume $(\mathbf{x}\mathbf{a}\mathbf{G} \Phi_{x_a})$. If $\mathbf{a}\mathbf{e}\mathbf{q} \mathbf{M} \mathbf{N} \in \Phi_{x_a}$, then $\mathbf{M} = \mathbf{N}$.

Theorem 10 (Admissibility of Symmetry and Transitivity)

Assume $(\mathbf{x}\mathbf{a}\mathbf{G} \Phi_{x_a})$.

1. If $[\Phi_{x_a} \mid - \mathbf{a}\mathbf{e}\mathbf{q} \mathbf{M} \mathbf{N}]$ then $[\Phi_{x_a} \mid - \mathbf{a}\mathbf{e}\mathbf{q} \mathbf{N} \mathbf{M}]$
2. If $[\Phi_{x_a} \mid - \mathbf{a}\mathbf{e}\mathbf{q} \mathbf{M} \mathbf{L}]$ and $[\Phi_{x_a} \mid - \mathbf{a}\mathbf{e}\mathbf{q} \mathbf{L} \mathbf{N}]$ then $[\Phi_{x_a} \mid - \mathbf{a}\mathbf{e}\mathbf{q} \mathbf{M} \mathbf{N}]$.

Linear Context Extensions with Alternative Declarations We extend here (Lemmas and Theorems 11–17) reflexivity, symmetry, and transitivity of algorithmic equality to the polymorphic lambda-calculus. See Appendix A.2 and Felty et al (2014, Section 3.2).

Theorem 11 (Admissibility of Reflexivity for Types, G Version)

Assume $(\mathbf{a}\mathbf{t}\mathbf{p}\mathbf{G} \Phi_{atp})$. If $[\Phi_{atp} \mid - \mathbf{is_tp} \mathbf{A}]$ then $[\Phi_{atp} \mid - \mathbf{a}\mathbf{t}\mathbf{p} \mathbf{A} \mathbf{A}]$.

Lemma 12 (G-Promotion for Type Reflexivity)

Assume $(\mathbf{a}\mathbf{e}\mathbf{q}\mathbf{G} \Phi_{aeq})$. If $[\Phi_{aeq} \mid - \mathbf{is_tp} \mathbf{A}]$ then $[\Phi_{aeq} \mid - \mathbf{a}\mathbf{t}\mathbf{p} \mathbf{A} \mathbf{A}]$.

Theorem 13 (Admissibility of Reflexivity for Terms, G Version)

Assume $(\mathbf{a}\mathbf{e}\mathbf{q}\mathbf{G} \Phi_{aeq})$. If $[\Phi_{aeq} \mid - \mathbf{is_tm} \mathbf{M}]$ then $[\Phi_{aeq} \mid - \mathbf{a}\mathbf{e}\mathbf{q} \mathbf{M} \mathbf{M}]$.

Theorem 14 (Admissibility of Reflexivity for Types, R Version)

Assume $(\mathbf{a}\mathbf{t}\mathbf{p}\mathbf{R} \Phi_\alpha \Phi_{atp})$. If $[\Phi_\alpha \mid - \mathbf{is_tp} \mathbf{A}]$ then $[\Phi_{atp} \mid - \mathbf{a}\mathbf{t}\mathbf{p} \mathbf{A} \mathbf{A}]$.

Lemma 15 (Relational Strengthening) Assume $(\mathbf{a}\mathbf{e}\mathbf{q}\mathbf{R} \Phi_{\alpha x} \Phi_{aeq})$. Then there exist contexts Φ_α and Φ_{atp} such that $(\mathbf{a}\mathbf{l}\mathbf{p}\mathbf{h}\mathbf{x}\mathbf{R} \Phi_{\alpha x} \Phi_\alpha)$, $(\mathbf{a}\mathbf{e}\mathbf{q}\mathbf{a}\mathbf{t}\mathbf{p}\mathbf{R} \Phi_{aeq} \Phi_{atp})$, and $(\mathbf{a}\mathbf{t}\mathbf{p}\mathbf{R} \Phi_\alpha \Phi_{atp})$.

Lemma 16 (R-Promotion for Type Reflexivity)

Assume $(\mathbf{a}\mathbf{e}\mathbf{q}\mathbf{R} \Phi_{\alpha x} \Phi_{aeq})$. If $[\Phi_{\alpha x} \mid - \mathbf{is_tp} \mathbf{A}]$ then $[\Phi_{aeq} \mid - \mathbf{a}\mathbf{t}\mathbf{p} \mathbf{A} \mathbf{A}]$.

Theorem 17 (Admissibility of Reflexivity for Terms, R Version)

Assume $(\mathbf{a}\mathbf{e}\mathbf{q}\mathbf{R} \Phi_{\alpha x} \Phi_{aeq})$. If $[\Phi_{\alpha x} \mid - \mathbf{is_tm} \mathbf{M}]$ then $[\Phi_{aeq} \mid - \mathbf{a}\mathbf{e}\mathbf{q} \mathbf{M} \mathbf{M}]$.

Non-Linear Context Extensions Here (Lemmas and Theorems 18–22), we go back to the untyped lambda-calculus, introduce declarative equality, and prove the completeness of algorithmic equality with respect to it. See Appendix A.1 and Felty et al (2014, Section 3.3).

Lemma 18 (G-Promotion for Reflexivity, Symmetry, and Transitivity)

Assume $(\text{daG } \Phi_{da})$.

1. If $[\Phi_{da} \mid - \text{is_tm } M]$ then $[\Phi_{da} \mid - \text{aeq } M \ M]$.
2. If $[\Phi_{da} \mid - \text{aeq } M \ N]$ then $[\Phi_{da} \mid - \text{aeq } N \ M]$.
3. If $[\Phi_{da} \mid - \text{aeq } M \ L]$ and $[\Phi_{da} \mid - \text{aeq } L \ N]$ then $[\Phi_{da} \mid - \text{aeq } M \ N]$.

Theorem 19 (Completeness, G Version)

Assume $(\text{daG } \Phi_{da})$. If $[\Phi_{da} \mid - \text{deq } M \ N]$ then $[\Phi_{da} \mid - \text{aeq } M \ N]$.

Lemma 20 (Relational Strengthening) Assume $(\text{daR } \Phi_{xa} \ \Phi_{xd})$. Then there exists a context Φ_x such that $(\text{xaR } \Phi_x \ \Phi_{xa})$.

Lemma 21 (R-Promotion for Reflexivity)

Assume $(\text{daR } \Phi_{xa} \ \Phi_{xd})$. If $[\Phi_{xd} \mid - \text{is_tm } M]$ then $[\Phi_{xa} \mid - \text{aeq } M \ M]$.

Theorem 22 (Completeness, R Version)

Assume $(\text{daR } \Phi_{xa} \ \Phi_{xd})$. If $[\Phi_{xd} \mid - \text{deq } M \ M]$ then $[\Phi_{xa} \mid - \text{aeq } M \ M]$.

Order The next benchmark considers order in a context using the same OL (the untyped lambda-calculus). See Felty et al (2014, Section 3.4).

Theorem 23 (Pairwise Substitution) Assume $(\text{xaG } \Phi_{xa})$.

If $[\Phi_{xa}, \text{block } (x:\text{tm}; \text{aeq } x \ x) \mid - \text{aeq } M1 \ M2]$ and $[\Phi_{xa} \mid - \text{aeq } N1 \ N2]$, then $[\Phi_{xa} \mid - \text{aeq } (M1 \ N1) \ (M2 \ N2)]$.

Uniqueness The next benchmark problem is type uniqueness of the simply-typed lambda-calculus. See Appendix A.3 and Felty et al (2014, Section 3.5).

Theorem 24 (Type Uniqueness)

Assume $(\text{xtG } \Phi_t)$. If $[\Phi_t \mid - \text{oft } M \ A]$ and $[\Phi_t \mid - \text{oft } M \ B]$, then $A = B$.

Substitution The final benchmark addresses the interaction of the substitution property with context reasoning. The OL is again the simply-typed lambda calculus, and judgments include a parallel reduction relation, which evaluates terms under lambda-abstractions. See Appendix A.4 and Felty et al (2014, Section 3.6).

Lemma 25 Assume $(\text{xtG } \Phi_t)$. If $[\Phi_t, \text{block } x:\text{tm}; \text{oft } x \ A \mid - \text{oft } (M \ x) \ B]$ and $[\Phi_t \mid - \text{oft } N \ A]$, then $[\Phi_t \mid - \text{oft } (M \ N) \ B]$.

Theorem 26 (Type Preservation for Parallel Reduction, R Version)

Assume $(\text{xrtR } \Phi_r \ \Phi_t)$. If $[\Phi_r \mid - \text{pr } M \ N]$ and $[\Phi_t \mid - \text{oft } M \ A]$, then $[\Phi_t \mid - \text{oft } N \ A]$.

Theorem 27 (Type Preservation for Parallel Reduction, G Version)

Assume $(\text{xrtG } \Gamma)$. If $[\Gamma \mid - \text{pr } M \ N]$ and $[\Gamma \mid - \text{oft } M \ A]$, then $[\Gamma \mid - \text{oft } N \ A]$.

When we refer to the “informal proofs” of the above lemmas and theorems in the rest of this paper, we mean the proofs described in Section 3 of the companion paper.

3 Mechanization in Twelf

Twelf supports specifying formal systems in the logical framework LF (Harper et al, 1993). LF is a very compact dependent type theory supporting atomic types a that can be indexed by terms M_i and (dependent) function types $\Pi x:A.B$. Following recent practice, we only characterize terms in normal form, since these are the only ones that are meaningful.

$$\begin{aligned} \text{Kind } K & ::= \text{type} \mid \Pi x:A.K \\ \text{Types } A, B & ::= a \ M_1 \dots M_n \mid \Pi x:A.B \\ \text{Terms } M, N & ::= x \ M_1 \dots M_n \mid c \ M_1 \dots M_n \mid \lambda x.M \end{aligned}$$

Logically, we can think of $a \ M_1 \dots M_n$ as a predicate. Implications correspond to the non-dependent function space $\Pi x:A.B$ where x does not occur in B . This is commonly abbreviated as $A \rightarrow B$. Universal quantification corresponds to the dependent function space $\Pi x:A.B$.

The grammar of LF does not separate between the types for terms and those for predicates. Rather, this distinction happens naturally on the level of kinds that classify types. A type family of level 0 has kind `type`. The type `tm` from the previous section is an example. A type family that depends on objects of level 0 such as `aeq` is a type of level 1, etc. Note the uniform treatment of binding structures. Not only are object-level binders on the level 0 modeled via a function space and hence naturally support object-level substitution via function application, but hypothetical and parametric derivations (i.e., objects of level 1) are characterized by a (dependent) function. Hence, substituting into a given derivation also boils down to function application. Since non-dependent functions are a special case of dependent functions, both the parametric and hypothetical substitution property collapses to the same substitution operation. As a consequence this methodology easily scales to encode and reason about *intrinsically* typed systems instead of stating the well-typedness relation separately. For a general introduction on how to encode object languages in LF, we recommend Pfenning (2001).

3.1 Basic Linear Context Extensions (G Version)

In Twelf proofs are implemented as a *relation* between derivations within the logical framework LF. For a summary of the specification of algorithmic equality, referred to below as `aeq M N`, we refer the kind reader to the appendix A.1. We begin by stating reflexivity of algorithmic equality (Theorem 8) as a relation `ref` between a term `M` and the algorithmic equality judgment `aeq M M`. It can be read as “For all `M:tm` there exists a derivation `aeq M M`. We implement the proof by analyzing and inducting `M` directly, since there is no essential distinction between terms (type family of level 0) and predicates (type families of level 1). Moreover we note that `ref` depends on `tm` (type family of level 0) and `aeq` (type family of level 1) and hence `ref1G` is a type family of level 2.

```
ref1G:  $\Pi M:tm. \text{aeq } M \ M \rightarrow \text{type}.$ 
```

```
r_a: ref1G M2 D2  $\rightarrow$  ref1G M1 D1
 $\rightarrow$  ref1G (app M1 M2) (ae_a D2 D1).
```

```
r_l: ( $\Pi x:tm. \Pi u:aeq \ x \ x. \Pi r_x: \text{ref1G } \ x \ u. \text{ref1G } (M \ x) \ (D \ x \ u)$ )
 $\rightarrow$  ref1G (lam  $\lambda x. M \ x$ ) (ae_l  $\lambda x. \lambda u. D \ x \ u$ )
```

The reflexivity proof is then implemented using two constants `r_a` and `r_l`. We can read `r_a` as follows: Given that `reflG M2 D2` (i.h. on `M2`) and `reflG M1 D1` (i.h. on `M1`), we can obtain a proof `reflG (app M1 M2) (ae_a D2 D1)`. The definition of `ae_a` and `ae_l` are given in the appendix A.1.

The case for lambda-terms is slightly more tricky. In Twelf, not only specifications but also proofs are represented in the implicit-context style. As a consequence the variable cases in a proof are treated for each variable that is introduced as we traverse a binder. Hence our recursive call (i.h.) will take place under the assumption that `x:tm`. Moreover, we cannot simply appeal to `reflG (M x) (D x u)`. First, we must know that `u:aeq x x`. Second, we must have that the reflexivity relation holds for the variable `x`. The base case, which in the on paper proof is handled separately, is folded into the case for lambda-abstraction. We can therefore read `r_l` as: Assuming that for all `x:tm` and `u:aeq x x`, `reflG x u` implies `reflG (M x) (D x u)`, we can establish that `reflG (lam λx.M x) (ae_l λx.λu.D x u)`.

LF type checking guarantees that all derivations that are manipulated and constructed are meaningful. The context of assumptions is ambient and implicit. However, for the relation `reflG` to be a proof, we also must establish that it implements a *total function*; in particular, we must guarantee that all the necessary base cases are taken care off. This cannot be achieved within the logical framework LF. Instead, Twelf relies on *external checkers*. There are three distinct properties we need to establish about the relation:

1. Well-modedness. The mode checker (Rohwedder and Pfenning, 1996) establishes that the given relation can be viewed as a non-deterministic function. In the given example, we specify that `M` is an input and `D` is the output of the relation using a mode declaration

```
%mode reflG +M -D.
```

Mode checking verifies that whenever the inputs are given the specified output can be computed; it does not, however, guarantee that there is a unique output value.

2. World satisfaction. The world checker verifies that the implemented relation introduces dynamic assumptions according to the predefined schema.

```
%block xaG_r : block {x:tm}{u:aeq x x}{r_x: reflG x u}.
%worlds (xaG_r) (reflG M D).
```

Thus, a context schema is formalized by a `%block` declaration and schema satisfaction by the `%worlds` check that verifies that indeed the reflexivity proof only introduces the specified dynamic assumptions. Twelf only permits the generalized contexts approach, since assumptions are ambient. Note that the schema defined differs from the schema definition `xaG` in the ORBI file, since in addition to `x:tm` and `u:aeq x x` the declaration also contains `reflG x u`. The defined schema of the eligible contexts is more specific and more restrictive than our definition in the companion paper. This is tied to the fact that we state proofs as relations using contexts implicitly. World checking is essential in Twelf to guarantee that all base cases are handled.

3. Totality. This checks two properties: termination and coverage. The termination checker (Pientka, 2005) verifies that the given relation is terminating according to a specified ordering, while the coverage checker verifies that a case exists for all possible inputs.

```
%total D (reflG M D).
```

We state that the relation `reflG` terminates because it is defined recursively on `D` and all appeals to the i.h. are structurally smaller.

The coverage checker (Schürmann and Pfenning, 2003) verifies that a given relation is covering, i.e., a case for all possible inputs is provided and splitting on an output argument does not neglect and drop possible cases.

Twelf’s meta-language to establish that a given relation constitutes a total function is restricted to the language \mathcal{M}^2 (Schürmann and Pfenning, 1998), a fragment of first-order logic which only admits $\forall - \exists$ formulas and does not allow arbitrary nesting of universal and existential quantification.

3.2 Linear Context Extensions with Alternative Declarations (G Version)

Extending the algorithmic equality case study to the polymorphic lambda-calculus is straightforward. For the specification of types, terms, algorithmic and declarative equality, we add the additional cases for handling type abstractions and applications (see Appendix A.2). Since types can occur in terms, we need to establish additional lemmas about the admissibility of reflexivity, symmetry and transitivity on the level of types. We only give the encoding of the reflexivity statement which reads as “for all `T:tp`, there exists a derivation `aeq T T` together with the meta-theoretic directives for mode, worlds, and totality. We note again that we implement the proof by recursion over `T:tp`.

```
reflTpG: //T:tp.atp T T → type.
%mode reflTpG +T -D.
%block atpG_r : block {a:tp}{u:atp a a}{r_a: reflTpG a u}.
%worlds (atpG_r) (reflTpG T D).
%total T (reflTpG T D).
```

Next, we extend the reflexivity proof `reflG` for terms with the additional cases for type abstraction and type application. We call the extended proof `reflTmG` to emphasize it considers terms only. There are two issues that arise: First, since the case for lambda-abstraction introduces assumptions about term variables and the equality of term variables, while the case for type abstraction introduces type variables and equality of type variables, we need to specify *alternating* assumptions in a world declaration. This alternation of blocks is described by the following world declaration `(xaG_r | atpG_r)`.

```
%block xaG_r : block {x:tm}{u:aeq x x}{r_x: reflTmG x u}.
%worlds (xaG_r | atpG_r) (reflTmG T D).
```

The admissibility of reflexivity is then established in the world (aka context) containing both blocks `xaG_r` and blocks `atpG_r`. The world checker then verifies that our context contains blocks of declarations consisting of assumptions `x:tm`, `u:aeq x x`, and `r_x: reflTmG x u` or assumptions `a:tp`, `u:atp a a`, and `r_a: reflTpG a u`.

Second, the reflexivity proof `reflTmG` for terms relies on the reflexivity proof `reflTpG` for types. This is justified by the fact that the world for `reflTpG` is a strict subset of the world specified for `reflTmG`, where the additional assumptions in block `xaG_r` are irrelevant to the reflexivity proof `reflTpG` for types. This is facilitated by

```

ceqG: deq M N → aeq M N → type.
c_r: reflG _ E → ceqG de_r E.
c_t: ceqG D1 E1 → ceqG D2 E2 → transG E1 E2 E → ceqG (de_t D2 D1) E.
c_s: ceqG D E1 → sym E1 E → ceqG (de_s D) E.
c_l: (Πx:tm. Πu:aeq x x. Πr_x:reflG x u. Πt_x:transG u u u. Πs_x:symG u u.
      Πv:deq x x. ceqG v u → ceqG (D x v) (E x u))
      → ceqG (de_l λx.λv.D x v) (ae_l λx.λu.E x u).
c_a: ceqG D1 E1 → ceqG D2 E2 → ceqG (de_a D1 D2) (ae_a E1 E2).
%mode ceqG +D -E.
%block daG_rtsc:block {x:tm}{u:aeq x x}
                    {r_x:ref x u}{t_x:tr u u u}{s_x:sym u u}
                    {v:deq x x}
                    {c_x: ceqG v u}.
%worlds (daG_rtsc) (ceqG D E).
%total D (ceqG D E).

```

Fig. 1 Completeness of Algorithmic Equality in Twelf.

Twelf’s *subordination* analysis, which maintains a dependency graph of all the type families. A type family a that is not a subordinate to a type family b (i.e., they do not depend on each other), can safely be added to (i.e., weakening) and removed from (i.e., strengthening) the context. Therefore, one might say subordination enables safe instances of context weakening and context strengthening (see Harper and Licata, 2007).

A similar issue arises in the transitivity and symmetry property for terms, which relies on the corresponding property for types.

3.3 Non-Linear Context Extensions (G Version)

We now inspect the implementation of the completeness proof of Theorem 19 given in Figure 1 which only considers equality on terms instead of polymorphic terms. We again define a type family `ceqG` describing the relation between `deq M N` and `aeq M N` (see Appendix A.1 for their specifications).

When considering each case in the proof, we need to pay attention to what implicit assumptions are needed. We consider each case for `deq T S` separately: If `de_r:deq M M` was used, then we appeal to the reflexivity lemma. We note that the reflexivity lemma is proven in a context containing `xaG_r`, namely `x:tm, u:aeq x x` and `r_x:reflG x u`. Therefore, we want the context for `ceqG` to contain at least these assumptions described in `xaG_r`. We say here “at least”, since the context in which we prove `ceqG` may contain more assumptions than needed by `reflG`, provided that such assumptions do not contribute to the proof of `reflG`. This is again enforced in Twelf by subordination analysis. If `de_t` was used, we have two sub-derivations: `D1:deq M L` and `D2:deq L N`. By induction, we obtain derivations `E1:aeq M L` and `E2:aeq L N`. To finish the proof, we appeal to the transitivity lemma. We note that the transitivity lemma was proven in a context containing the block `x:tm, u:aeq x x`, and `t_x:transG u u u`. The case for symmetry using `de_s` is similar. The implicit context for the completeness proof must therefore contain

the union of the assumptions needed for applying the reflexivity, transitivity, and the symmetry lemma.

The case for application is straightforward. Finally, we consider the case for lambda-abstractions, where we have used `de_1`, declarative equality for lambda-terms, and the context is extended. Intuitively, we want to introduce a variable `x` together with the assumption `v:deq x x` and `u:aeq x x`, and simply appeal to the induction hypothesis `ceqG (D x v) (E x u)`. However, since all the information pertaining variables is folded into the binder-case, we also need to add assumptions `r_x:reflG x u`, `t_x:transG u u u`, `s_x:symG u u` and `ceqG v u`; in other words, we must assume that the reflexivity, symmetry and transitivity lemma hold for the variable `x` in addition to assuming that indeed there is a proof `ceqG v u` that guarantees that whenever we have `deq x x` we must have a proof for `aeq x x`.

To verify that the implemented relation constitutes a proof, we again check well-modedness, world satisfaction, and totality. World checking will make sure that we can safely refer to the transitivity and reflexivity lemma which both are proven in a smaller context. To ensure that this check is reasonably efficient, Twelf verifies that all blocks occurring in the context of the lemma are a prefix of some block occurring in the context of the main theorem; in addition, the type families in the lemma (i.e., callee) is a subordinate to the additional type families in the theorem (i.e., caller). This guarantees that the additional type families in the theorem are irrelevant to the lemma.

This notion of *context subsumption* is a sufficient condition for weakening and strengthening the contexts associated with a given type family and obviates in many cases the need to prove weakening and strengthening lemmas. However, it can also be brittle, since the order of assumptions matters. For example, in the given example we have to weaken the implicit context in our proof for transitivity and include the hypothetical assumption `r_x:reflG x u`,¹ to take full advantage of context subsumption as it is presently implemented in Twelf. We also need to weaken the symmetry proof to include the assumption `r_x:reflG x u` and `t_x:transG u u`, since `block {x:tm}{u:aeq x x}{s_x:symG u u}` is not a prefix of the block `daG_rtsc`.

For a more detailed explanation regarding mechanizing proofs in the Twelf system and context subsumption, we refer the reader to Harper and Licata (2007).

3.4 Order (G Version)

To encode the pairwise substitution lemma 23 we cannot rely simply on LF function application. The code is reported in Fig. 2; specification of algorithmic equality, i.e., `aeq M N is` given in the Appendix A.1. Exchange is handled straightforwardly. However, an interesting issue arises in the base cases. We have one base case `s_x`, where `x` is the variable we want to replace and `u:aeq x x` describes the derivation that `x` is equal to itself. In this case we simply return the derivation `F`. The second base case arises when traversing a lambda-abstraction. Here we encounter a variable `y` together with `ae_y:aeq y y` and we want to return `aeq y y` no matter what we substitute for `x`. We hence add `IIN1:tm.IIN2:tm.IIF:aeq N1 N2.subst (λx.λae_x.ae_y)F ae_y` which can be read as: for all `N1,N2` where

¹ Alternatively, we can also weaken the reflexivity lemma and change the order of blocks.

```

subst: (IIx:tm.aeq x x → aeq (M1 x) (M2 x)) → aeq N1 N2
      → aeq (M1 N1) (M2 N2) → type.

s_x: subst (λxλu. u) F F.

s_l: (II y:tm.II ae_y:aeq y y.
      (IIN1:tm.IIN2:tm.IIF:aeq N1 N2.subst (λx.λae_x. ae_y) F ae_y) →
      subst (λx.λu. D x u y v) F (E y v) )
      → subst (λxλu.ae_1 λy.λv. D x u y v) F (ae_1 E ).

s_a: subst (λx.λu. D1 x u) F E1 → subst (λx.λu. D2 x u) F E2
      → subst (λx.λu. ae_a (D1 x u) (D2 x u)) F (ae_a E1 E2).

%mode subst +D +F -E.
%block xaG_s : block {y:tm}{ae_y:aeq y y}
  {s_y:IIN1:tm.IIN2:tm.IIF:aeq N1 N2. subst (λx.λae_x.ae_y) F ae_y}.
%worlds (xaG_s) (subst D F E).
%total D (subst D F E).

```

Fig. 2 Pairwise Substitution Lemma in Twelf

$F:aeq\ N1\ N2$, substituting F for x yields ae_y . This is an example of a higher-order assumptions that Twelf supports.

3.5 Uniqueness (G Version)

We consider next the implementation of the proof of type uniqueness for simply typed lambda-terms. The specification of terms together with their typing rules can be found in Appendix A.3. To express the relation between a term M and its corresponding type T , we write $oft\ M\ T$. Since Twelf has no built-in equality, we must define it. There is a choice to define equality simply using reflexivity, $e_ref: eq\ T\ T$, or recursively based on the structure of the type. This choice surprisingly matters in Twelf. We sketch the first approach in Fig. 3, by defining a relation `unique`. The implementation of the proof follows earlier ideas with one exception. We cannot simply refer to `e_ref`. The reason has to do with type reconstruction, which will consider the inputs and outputs of a given relation. Let's for example consider the case (denoted by `u_1`) where we have used the typing rule for lambda-terms. When we write `e_ref` in the output position in clause `u_1`, we will restrict what T and T' can be – effectively, we are setting them to be identical. This will cause coverage to fail (correctly), because there is in fact no case where the types are different. Relations unlike functions cannot always capture the flow and refinement of information that happen in the proof. One solution is to prove some inversion lemmas for function types and then appeal to them, but avoid unduly restricting arguments in the main theorem. We again rely on subordination to justify calling the lemma `arr_inv1` and `arr_inv2` that are defined in the empty context (world) within the context `xtG_u`. Finally, we remark that to avoid adding for each individual variable the base case `unique u u e_ref`, we could omit it and add instead a generic base case `u_b:unique D D e_ref`. This would allow us to define a tighter context only consisting of `x:tm` and `d:oft x T`.

A different approach is to define equality on the structure of the type as a judgment `atp T T` and then prove that reflexivity is admissible without further

```

% Inversion lemmas about function types
arr_inv1 : eq (arr T1 T2) (arr T1' T2') → eq T2 T2' → type.
arr_inv1_ref : arr_inv1 e_ref e_ref.
...
arr_inv2 :  $\Pi T:tp. eq T2 T2' \rightarrow eq (arr T T2) (arr T T2') \rightarrow type.$ 
arr_inv2_ref : arr_inv2 T e_ref e_ref.
%mode arr_inv2 +T +E1 -E2.
%worlds () (arr_inv2 T E1 E2).
%total E1 (arr_inv2 _ E1 _).

% Type uniqueness proof
unique : oftM T → oftM T' → eq T T' → type.
u_a : arr_inv1 E E' → unique D1 F1 E
      → unique (of_a D1 D2) (of_a F1 F2) E'.
u_l : arr_inv2 T1 E E'
      → ( $\Pi x:tm. \Pi u:oft x T1. unique u u e\_ref \rightarrow unique (D x u) (F x u) E$ )
      → unique (of_l  $\lambda x. \lambda u. D x u$ ) (of_l  $\lambda x. \lambda u. F x u$ ) E'.

%mode unique +D +F -E.
%block xtG_u : some {T:tp} block {x:tm} {d:oft x T} {u:unique d d e_ref}.
%worlds (xtG_u) (unique D F E).
%total D (unique D _ _).

```

Fig. 3 Type Uniqueness in Twelf

ado, avoiding the previous coverage problems, but reasoning structurally about equality.

3.6 Substitution

Twelf models hypothetical and parametric derivations as LF functions. Therefore, substitution can be obtained for free relying on LF function application. This is illustrated in the type preservation proof for parallel reductions. Parallel reductions, described by $pr M N$, together with typing rules, described by $oft N A$, are specified in the Appendix A.4. We define the theorem 23, which states that if $pr M N$ and $oft M A$, then $oft N A$, and its corresponding proof below.

```

tps : pr M N → oftM A → oftN A → type.
%mode tps +R +D -F.

tp_b: tps (pr_b R1 R2) (of_app (of_lam D1) D2) (F1 N F2)
      ← ( $\Pi x:tm. \Pi of_v:oft x A. \Pi pr_v:pr x x. tps pr_v of_v of_v$ )
      → tps (R1 x pr_v) (D1 x of_v) (F1 x of_v)
      ← tps R2 D2 F2.

%block xrtG_tpv : some {t:tp} block {x:tm}{of_v: oft x t}{pr_v: pr x x}
      {tp_v:tps pr_v of_v of_v}.
%worlds (xrtG_tpv) (tps R D F).
%total R (tps R D F).

```

We want to draw particular attention to the case tp_b which considers the case for beta-reduction. The remaining cases can be found in the online appendix. We consider the reduction of $(\lambda x.M x) N$ (rule pr_b in the appendix A.4). $R1$ denotes a reduction sequence for $pr (M x) (M' x)$, given that $pr x x. R2$ describes

the reduction sequence for $\text{pr } N \ N'$. By the i.h. on R1, we obtain a derivation $F1: \Pi x:\text{tm} . \text{oft } x \ A \ \rightarrow \text{oft } (M \ x)B$. Moreover, by the i.h. on R2, we obtain a derivation $F2: \text{oft } N \ A$. We now want to substitute N for x in $F1$ and subsequently $F2$ for the assumption $\text{oft } N \ A$ in $F1$. Since $F1$ denotes a function, we simply use function application to model substitution writing $(F1 \ N \ F2)$.

4 Mechanization in Beluga

Beluga is a proof and programming environment (Pientka, 2008; Pientka and Dunfield, 2008, 2010) that also supports the specification of formal systems in the logical framework LF. On top of LF Beluga provides a dependently typed functional language that allows programmers to manipulate and analyze contexts and contextual LF objects, i.e., a LF object M in a context of assumptions Ψ (Nanevski et al, 2008), which we write as $[\Psi.M]$. The dependency on a context is also captured on the level of types via the contextual type $[\Psi.A]$. We say a contextual object $[\Psi.M]$ has type $[\Psi.A]$ if M has type A in the context Ψ . This allows us to concisely represent and manipulate hypothetical and parametric derivations, i.e., a goal formula (object) together with the assumptions that we are allowed to use to establish the goal.

Beluga’s type language takes contextual types as base types; it also supports product types $(T_1 * T_2)$ and function types $(T_1 \rightarrow T_2)$, and we can universally quantify over contextual objects and contexts $(\{X:U\}T$ where U is either a contextual type $[\Psi.A]$ or describes a context schema, i.e., the type of a context). Logically, Beluga’s type language corresponds to first-order logic where we quantify over contextual objects and contexts and take contextual objects as our basic predicates.

Beluga’s computation language supports tuples (introduction for product types), nameless function $(\text{fn } x \Rightarrow e)$ (introduction for function types), abstraction over contextual objects and contexts $(\lambda^{\square} X \Rightarrow e)$ (introduction for universal types), recursion $(\text{rec } f \Rightarrow e)$, and case-analysis over contextual objects and contexts. Unlike Twelf’s meta-logic that prevents nesting of quantifiers and implications, Beluga’s computation language imposes no such restrictions and supports higher-order functions. In this Section, we will concentrate on showing how to implement inductive proofs from our benchmark using generalized approach; we also briefly discuss how to encode proofs in the relational approach.

Inductive proofs in Beluga are represented as recursive functions about contextual LF objects using pattern matching. Each case of the proof corresponds to one branch in the function. Most recently Cave and Pientka have shown how to extend Beluga with support for inductive definitions (i.e., recursive datatypes) (Cave and Pientka, 2012) and first-class substitutions (Cave and Pientka, 2013) leading to Beluga^μ. This allows us to define predicates about contexts and contextual objects, relate contexts via substitutions, and more importantly prove properties about them inductively.

4.1 Basic Linear Context Extensions (G Version)

We consider first the proof that reflexivity is admissible for algorithmic equality (Theorem 8). For a summary of the specification of algorithmic equality, written

```

rec reflG : {γ:xaG}{M:[γ.tm]} [γ. aeq (M..) (M..)] =
λ□γ ⇒ λ□M ⇒ case [γ. M..] of
| [γ. #p.1 ..] ⇒ [γ. #p.2 ..]           % Variable
| [γ. lam λx. M.. x] ⇒                 % Lambda
  let [γ,b:block y:tm, ae_v:aeq y y. D.. b.1 b.2]=
      reflG [γ, b:block y:tm, ae_v:aeq y y] [γ, b. M.. b.1]
  in [γ. ae_1 λx. λw. (D.. x w)]
| [γ. app (M1..) (M2..)] ⇒             % Application
  let [γ. D1..] = reflG [γ] [γ . M1.. ] in
  let [γ. D2..] = reflG [γ] [γ . M2.. ] in
  [γ. ae_a (D1..) (D2..)];

```

Fig. 4 Reflexivity of Algorithmic Equality in Beluga

as $\text{aeq } M \ N$, we refer the kind reader to the appendix A.1. Just as types classify expressions, contexts are classified by context schemas and the definition of schemas in the ORBI spec directly translates to Beluga.²

```

schema xaG = block x:tm, u:aeq x x;

```

The schema states that our context consists of a block of assumptions, containing $x:\text{tm}$ and $\text{aeq } x \ x$. More formally, the block-construct introduces a Σ -type grouping the two declarations together. Schemas are closely related to the world declarations in Twelf. However, there is a crucial difference. In Twelf a world declaration also contains the base cases in a proof; for example, in the reflexivity proof, the world declaration also contained $\text{ref } x \ u$, a proof that the statement holds for each x we encounter. In Beluga it is sufficient to keep track of $x:\text{tm}$ and $u:\text{aeq } x \ x$, as we are abstracting over the context and will be able to write a generic variable case covering the scenario where we consider a variable from the context. Moreover, schema checking in Beluga unlike world checking in Twelf does not check that a given type family satisfies the declared schema; it merely ensures that contexts that are constructed and passed as arguments in computations satisfy the declared schema.

The reflexivity (Theorem 8) can be implemented as a recursive function called `reflG` of type: $\{\gamma:\text{xaG}\}\{M:[\gamma.\text{tm}]\}[\gamma.\text{aeq } (M..)(M..)]$ (see Fig. 4). This can be read as: for all contexts γ that have schema xaG , for all terms M , we have a proof that $[\gamma. \text{aeq } (M..)(M..)]$. Quantification over contexts and contextual objects in computation-level types is denoted by curly braces; the corresponding abstraction on the level of expressions is written as $\lambda^{\square}\gamma \Rightarrow \lambda^{\square}M \Rightarrow e$.

Central is the idea of a contextual type. For example, M has type $[\gamma.\text{tm}]$, which describes an object M that has type tm in the context γ . Hence, M is a term that may refer to variables in the context γ . When we use M it is associated with a substitution that maps all the variables in γ to the correct target context. In the example, we use M within the contextual type $[\gamma.\text{aeq } (M..)(M..)]$. Hence, M is declared in the context γ and because it is also used in the context γ , it is associated with the identity substitution, which is written as $..$ in our concrete syntax. Intuitively, it means that M can depend on all the variables that occur in the context described by γ . The derivation $\gamma \vdash \text{aeq } M \ M$ is directly captured by

² Beluga's syntax for schemas differs slightly from ORBI's syntax.

the contextual type $[\gamma.\text{aeq } (M..)(M..)]$. While the informal on paper definitions often leave implicit the question in what context a term is well-formed, Beluga’s concrete syntax forces the programmer to be more precise.

In the proof for `ref1G` we begin by introducing γ and M followed by a case analysis on $[\gamma.M..]$ using pattern matching. There are three possible cases for M :

1. Variable case. If M is a variable from γ , we write $[\gamma.\#p.1..]$ where $\#p$ denotes a parameter variable declared in the context γ . Operationally, $\#p$ can be instantiated with any bound variable from the context γ . Since the context γ has schema `xaG`, it contains blocks `x:tm`, `ae_v:aeq x x`. The first projection of it allows us to extract the term component, while the second projection of it denotes the proof of `aeq x x`.
2. Lambda case. If M is a lambda-term, then we extend the context and appeal to the induction hypothesis by making a recursive call. Beluga supports declaration weakening as described in the companion paper (Felty et al, 2014), which allows us to use M that has type $[\gamma, x:\text{tm} . \text{tm}]$ in the extended context $[\gamma, b:\text{block } x:\text{tm}, \text{ae}_v: \text{aeq } x x]$. We simply construct a weakening substitution `b.1` with domain $\gamma, x:\text{tm}$ and range $\gamma, b:\text{block } x:\text{tm}, \text{ae}_v:\text{aeq } x x$ that essentially renames x to `b.1` in M . The recursive call returns $[\gamma, b:\text{block } y:\text{tm}, \text{ae}_v:\text{aeq } y y, D..b.1 b.2]$. Using it together with rule `ae_1` we build the final derivation.
3. Application case. If M is an application, we appeal twice to the induction hypothesis and build a proof for $[\gamma.\text{aeq } (\text{app } (M1..) (M2..)) (\text{app } (M1..) (M2..))]$.

Beluga’s type checker verifies that the manipulated derivations are well-formed and meaningful. For the recursive function `ref1G` to constitute a proof, we also need to know that it is total, i.e., all cases are covered and the function is terminating. At the time of writing, Beluga provides a coverage checker that will automatically verify whether all cases are covered (Dunfield and Pientka, 2009); the fact that a given function is terminating must be verified manually by the programmer. However, such arguments are usually direct and straightforward (see also Pientka et al, 2014, for a foundation restricting Beluga’s language to primitive structural recursive functions).

4.2 Context Extensions with Alternative Declarations (G Version)

Extending the algorithmic equality case study to the polymorphic lambda-calculus to handle alternating assumptions is straightforward. For the specification of the algorithmic equality rules for polymorphic terms (written as `aeq M N`) and types (written as `atp T S`), we refer the kind reader the Appendix A.2. We can now establish reflexivity, transitivity, and symmetry of type equality in a straightforward way. These proofs are established in the context of schema `atpG` with

```
schema atpG = block (a:tp, v:atp a a);
```

Then, we add the additional case for type abstraction and type application to reflexivity, transitivity and symmetry of term equality.

Similar issues as in the Twelf development arise; however the solution taken differs slightly.

1. Since the case for lambda-abstraction deals with term assumptions while the type abstraction introduces type assumptions, we need to specify *alternating* assumptions. This alternation of blocks is described by using `+` in Beluga's concrete syntax.

```
schema aeqG = block (x:tm , u:aeq x x) + block (a:tp , v:atp a a);
```

2. The reflexivity, transitivity and symmetry property for terms relies on the corresponding property for types. As in Twelf, Beluga's type inference engine relies on subordination to justify instances of safe context weakening and strengthening.³ When we call a function that requires a context γ of schema `atpG` but we have a context of schema `aeqG`, we check that every block `b` in schema `atpG` is a prefix of a block `b'` in the schema `aeqG` and moreover additional declarations present in `b'` are irrelevant to the declarations in `b`. The type checker therefore allows us to pass a context of schema `aeqG` whenever a context `atpG` is required.

The main difference between Twelf and Beluga with respect to the handling of contexts is which assumptions we track. Since Beluga is a functional language and allows users to implement a general variable case, the context characterizes only assumptions arising from the derivations we manipulate and matches our informal description given earlier. As a consequence, context subsumption often can avoid the difficulties that arise in Twelf due to the additional assumptions dealing with base cases.

4.3 Non-Linear Context Extensions (G Version)

We implement the completeness proof (theorem 19) as as a recursive function `ceqG`. We first define the schema of the generalized context, following our informal development as follows:

```
schema daG = block (x:tm , u:aeq x x , v:deq x x) ;
```

As in Twelf, the order in the schema matters, since context weakening and strengthening, as implemented in Beluga, will verify that every block in `schema xaG = block x:tm , u:aeq x x` is a prefix of a block in the context of schema `daG`. The observant reader will notice that our definition of `daG` does not match precisely the definition given in the ORBI file for Thm. 19, but we reordered the assumption `aeq x x` and `deq x x` to enable context subsumption. As in Twelf, Beluga will verify whether the context required by the lemma is a proper sub-context of the context provided by the main theorem. In the type and implementation of the function `ceqG`, we indicate that the context γ is implicit in the actual implementation of the proof and will be reconstructed by using round brackets and writing the context declaration as `(γ :daG)`.

We explain here the three cases shown in the informal proof in the companion paper (Felty et al, 2014). First, let us consider the case where we used an assumption from the context. Since the context γ consists of blocks with the following structure: `block x:tm , ae_v:aeq x x , de_v: deq x x`, we in fact want to match on

³ However, the operational semantics is not adapted and these programs are not executable; this would require us to add appropriate casting operations, which correspond to strengthening and weakening functions.

```

rec ceqG: (γ:daG) [γ. deq (T..) (S..)] → [γ.aeq (T..) (S..)] =
fn e ⇒ case e of
| [γ . #p.3..] ⇒ [γ . #p.2..]           % Assumption from context
| [γ. de_r ] ⇒ reflG [γ] [γ. _ ]       % Reflexivity
| [γ . de_t (D1..) (D2..)] ⇒           % Transitivity
  let [γ. F2..] = ceqG [γ. D2..] in
  let [γ. F1..] = ceqG [γ. D1..] in
  trans [γ. F1..] [γ. F2..]
| [γ. de_s (D..)] ⇒                     % Symmetry
  let [γ. F..] = ceqG [γ. D..] in
  sym [γ.F..]
| [γ. de_l (λx. λu. D.. x u)] ⇒         % Abstraction
  let [γ,b:block (x:tm,ae_v:aeq x x, de_v: deq x x). F.. b.1 b.2] =
    ceqG [γ, b:block (x:tm, ae_v:aeq x x, de_v: deq x x) . D.. b.1 b.3]
  in
  [γ. ae_lam (λx.λv. F.. x v)]
| [γ. de_a (D2..) (D1..)] ⇒           % Application
  let [γ. F1..] = ceqG [γ. D1..] in
  let [γ. F2..] = ceqG [γ. D2..] in
  [γ. ae_a (F1..) (F2..)] ;

```

Fig. 5 Completeness of Algorithmic Equality in Beluga

the third element of such a block. This is written as `#p.3 ...`. The type of `#p.3` is `deq (#p.1 ..) (#p.1 ..)`. Since our context always contains a block and the parameter variable `#p..` describes such a block, we know that there exists a proof for `aeq (#p.1 ..) (#p.1 ..)`, which can be described by `#p.2 ...`

Second, we consider the case where we applied the reflexivity rule `de_r` as a last step. In this case, we need to refer to the reflexivity lemma we proved about algorithmic equality. To use the function `reflG`, which implements the reflexivity lemma for algorithmic equality, we need a context of schema `xaG`; however, the context used in the proof for `ceqG` is of schema `daG` and we rely on context subsumption to justify passing a context `daG` in place of a context `xaG`. The cases for transitivity and symmetry are similar.

Third, we consider the case for `de_l`, the case for lambda-abstractions. In this case, we extend the context with the new declarations about variables and pass to the recursive call `ceqG` the derivation `[γ, b:block (x:tm,ae_v:aeq x x, de_v: deq x x). D..b.1 b.3]`. Declaration weakening (in the informal proof *d-wk* (Felty et al, 2014)) is built-in. In the pattern, the derivation `D` has type `[γ, x:tm, ae_v:aeq x x . deq (M..x) (N.. x)]`. We hence construct a weakening substitution `.. b.1 b.3` that allows us to move `D` to the context `γ, b:block (x:tm,ae_v:aeq x x, de_v:deq x x)`. The result of recursive call is a derivation `F`, where `F` only depends on `x:tm` and `u:aeq x x`. In the on-paper proof we refer to declaration strengthening (*d-str*) to justify that `F` cannot depend on `de_v` assumptions. In Beluga, the programmer uses strengthening by stating which assumptions `F` can depend on. The coverage checker will then subsequently rely on subordination to verify that the restricted case is sufficient and no other cases have been forgotten. Subordination allows us to verify that indeed assumptions of type `de_v: deq x x`

```

schema xaG = block x:tm, ae_v:aeq x x;
rec subst: (g:xaG)
  [γ, b:block (x:tm, ae_v:aeq x x).aeq (M1.. b.1) (M2.. b.1)]
  → [γ. aeq (N1..) (N2..)]
  → [γ. aeq (M1.. (N1..)) (M2.. (N2..)) ] =
fn d ⇒ fn d' ⇒ case d of
| [γ, b:block (x:tm, ae_v:aeq x x) . b.2] ⇒ d'
| [γ, b:block (x:tm, ae_v:aeq x x) . #p.2.. ] ⇒ [γ. #p.2..]
| [γ, b:block (x:tm, ae_x:aeq x x) . ae_1 λy.λae_y. D.. b y ae_y ] ⇒
  let [γ . D' .. ] = d' in
  let [γ, d': block (y:tm, ae_y: aeq y y) . F.. d'.1 d'.2] =
    subst [γ, d': block (y:tm, ae_y: aeq y y),
           d : block (x:tm, ae_x:aeq x x) . D.. d d'.1 d'.2 ]
        [γ, d' . D' .. ] in
    [γ. ae_1 λy.λae_y. F.. y ae_y]
| [γ, b:block (x:tm, ae_x:aeq x x) . ae_a (D1.. b) (D2.. b)] ⇒
  let [γ.F1..] = subst [γ, b:block(x:tm, ae_x:aeq x x). D1.. b] d' in
  let [γ.F2..] = subst [γ, b:block(x:tm, ae_x:aeq x x). D2.. b] d' in
  [γ. ae_a (F1..) (F2..)];

```

Fig. 6 Pairwise Substitution Lemma in Beluga

cannot be used in proofs for $\text{aeq } (M.. b.1) (N.. b.1)$. Finally, we use F to assemble the final result $\text{ae}_1 (\lambda x.\lambda v. F..x v)$.

We conclude this example with a few observations: The statement of the theorem is directly and succinctly represented in Beluga using contextual types and contextual objects. Every case in the on-paper proof corresponds directly to a case in the implementation of the recursive function. Type reconstruction is used to reconstruct implicit type arguments and infer the type of free contextual variables that occur in patterns. This is crucial to achieve a palatable source language. Weakening and strengthening are supported in Beluga through the typing rules and on the level of context variables and context schemas using context subsumption. If schema W is a prefix of a schema W' , then we can always use a context of schema W' in place of a context of schema W .

4.4 Order (G Version)

The order of assumptions in a context is important in Beluga. However, sometimes the need to reorder assumptions arises, as is illustrated in the proof of the substitution lemma for algorithmic equality. As in Twelf this kind of proof (Theorem 23) does not come for free in Beluga (see Fig. 6).

In Beluga, exchanging the order of assumptions in the context is managed via substitutions. In the case ae_1 , we consider $D.. b y \text{ae}_y$ where D is a derivation in the context $\gamma, b:\text{block } (x:\text{tm}, \text{ae}_x:\text{aeq } x x), y:\text{tm}, \text{ae}_y:\text{aeq } y y)$. To appeal to the i.h., we make the recursive call on $D.. d d'.1 d'.2$ in the context $\gamma, d':\text{block } (y:\text{tm}, \text{ae}_y:\text{aeq } y y), d:\text{block } (x:\text{tm}, \text{ae}_x:\text{aeq } x x)$. The substitution $.. d d'.1 d'.2$ provides a mapping from the variables in the context $\gamma, b:\text{block } (x:\text{tm}, \text{ae}_x:\text{aeq } x x), y:\text{tm}, \text{ae}_y:\text{aeq } y y$ to the variables in $\gamma, d':\text{block } (y:$

```

schema xtG = some [t:tp] block x:tm , u:oft x t;
rec unique : (γ:xtG)[γ.oft (M..) T] → [γ.oft (M..) S] → [.eq T S] =
fn d ⇒ fn f ⇒ case d of
| [γ. of_app (D1..) (D2..)] ⇒
  let [γ. of_app (F1..) (F2..)] = f in
  let [. e_ref] = unique [γ. D1..] [γ. F1..] in
  [. e_ref]
| [γ. of_lam (λx.λu. D.. x u)] ⇒
  let [γ. of_lam (λx.λu. F.. x u)] = f in
  let [. e_ref] = unique [γ,b: block x:tm , t:oft x _ . D.. b.1 b.2]
    [γ,b. F.. b.1 b.2] in
  [. e_ref]
| [γ. #p.2..] ⇒
  % d : oft #p.1 T
  let [γ. #q.2..] = f in % f : oft #p.1 S
  [. e_ref];

```

Fig. 7 Type Uniqueness in Beluga

tm , $\text{ae}_y:\text{aeq } y \ y$), $\text{d}:\text{block } (x:\text{tm}, \text{ae}_x:\text{aeq } x \ x)$. It is therefore a permuting substitution.

This proof also highlights another interesting aspect of Beluga: it allows recursion over the context. There are two base cases in this proof. First, we consider the derivation $\text{aeq } x \ x$ where we want to replace the variable x by the term $N1$ in $M1$ and by $N2$ in $M2$. Since we have a derivation d' describing $[\gamma.\text{aeq } (N1 \ ..) (N2 \ ..)]$ by assumption, we simply return d' .

Second, we want to consider the case where we have a variable from the context, but this variable is not x as in $[\gamma,b:\text{block } x:\text{tm}, \text{ae}_v:\text{aeq } x \ x. \ #p.2..]$. We know that $\#p..$ describes some assumption from γ and in particular, x cannot occur in it, since otherwise the context would be ill-formed. The substitution \cdot associated with $\#p.2$ is a (weakening) substitution with domain g and range γ , $b:\text{block } x:\text{tm}, \text{ae}_v:\text{aeq } x \ x$. We can therefore simply return $[\gamma. \ #p.2..]$.

4.5 Uniqueness (G Version)

We discuss next the implementation of the proof of type uniqueness for simply typed lambda-terms. The specification of terms together with their typing rules can be found in Appendix A.3. To express the relation between a term M and its corresponding type T , we write $\text{oft } M \ T$. Since Beluga does not support equality types, we implement equality using a LF type family eq as we did in the Twelf section. The uniqueness proof (see Fig. 7) itself is mostly straightforward (see also Pientka, 2010, for a longer discussion). The only interesting case arises when we encounter a variable of type tm from the context. Since our context consists of blocks containing variables of type tm and assumptions $\text{oft } x \ T1$, we pattern match on $[g.\#p.2..]$, i.e., we project out the second argument of the block. By the given assumptions, we know that $[g.\#p.2..]$ has type $[g.\text{oft } (\#p.1 \ ..)T]$, because $\#p$ has type $[g. \ \text{block } x:\text{tm}, u:\text{oft } x \ T]$. We also know that the second input, called f , to the function unique has type $[g. \ \text{oft } (\#p.1 \ ..)S]$. By inversion on f , we know

```

schema xrtG = some [t:tp] block x:tm, of_x: oft x t, pr_x: pr x x;
rec tps:( $\gamma$ :xrtG)[ $\gamma$ .pr (M..) (N..)]  $\rightarrow$  [ $\gamma$ .oft (M..) A]  $\rightarrow$  [ $\gamma$ .oft (N..) A] =
fn f  $\Rightarrow$  fn d  $\Rightarrow$  case f of
| [g. #p.3.]  $\Rightarrow$  d
| [g. pr_b ( $\lambda x$ . $\lambda$ pr_x. R1.. x pr_x) (R2.. )]  $\Rightarrow$ 
  let [g. of_a (of_l ( $\lambda x$ . $\lambda$ of_x. D1.. x of_x)) (D2..)] = d in
  let [g, b:block (x:tm, of_x: oft x T, pr_v: pr x x) . F1.. b.1 b.2] =
    tps [g, b:block (x:tm, of_x:oft x _, pr_v:pr x x) . R1.. b.1 b.3]
      [g, b . D1.. b.1 b.2] in
  let [g . F2..] = tps [g. R2..] [g. D2..] in
    [g . F1.. _ (F2..)] % use substitution lemma directly

```

Fig. 8 Type Preservation for Parallel Reductions in Beluga

that the only possible object that can inhabit this type is `[g.#p.2..]` and therefore `S` must be identical to `T`. Moreover, `#q` denotes the same block as `#p`.

We note that the difficulties of defining equality via reflexivity in Twelf do not arise in Beluga; Beluga’s proof is implemented as a function using pattern matching instead of relations; as we pattern match we learn more about the given derivations and information flows as expected.

4.6 Substitution (G Version)

Beluga enjoys the usual substitution property for parametric and hypothetical derivations for free. Consider the proof of type preservation for the simply-typed lambda-calculus with parallel reductions (Theorem 26): when `M` steps to `N` and `M` has type `A` then `N` has the same type `A`. Since we allow reductions underneath an abstraction, we may step terms containing variables. We therefore allow expressions to depend on the context γ . We only show the case for beta-reduction in Figure 8. Consider the clause `pr_b: pr (app (lam λx .M..x) (N..)) (M.. (N..))`. Then we have as assumption `d: [γ .oft (app (lam A (λx . M..x))(N..)) (arr A B)]`. By inversion, we know that `d: [γ . of_a (of_l λx . λu . D1..x u) (D2..)]` where `D1` stands for `oft (M..x) B` in the extended context γ , `x:tm`, `u:oft x A`. We also know that `D2` describes a derivation `oft (N..) A`. By the i.h. (recursive call) on `D2`, we therefore know `F2:oft (N'..) A`. By the i.h. (recursive call) on `D1`, we obtain a derivation `F1:oft (M'..x) B` in γ , `b:block (x:tm, of_x:oft x A)`. We now want to substitute for `x` the term `M'`, and for the derivation `oft x A` the derivation `F2`. This is achieved by applying to `F1` the substitution `.. _ (F2..)`. Since in the program above we do not have the name `N` available, we write an underscore and let Beluga’s type reconstruction algorithm infer the appropriate name.

4.7 Mechanizing Properties in Beluga^μ using Recursive Types and Substitutions

So far, we have concentrated on mechanizing proofs in Beluga only relying on a fragment which contains (dependent) functions and applications, and pattern matching on contextual objects and contexts. However, Beluga^μ (Cave and Pientka, 2012) also supports indexed recursive datatypes and first-class substitutions

(Cave and Pientka, 2013). This allows us to define predicates about contexts and contextual objects, relate contexts via substitutions, and more importantly prove properties about them inductively. The additional expressiveness gained allows us to express explicitly promotion lemmas and proofs using context relations (R Version).

4.7.1 Promotion

In the previous sections, we have taken advantage of built-in context subsumption in checking Beluga proofs. If schema W is a prefix of a schema W' , then we can always use a context of schema W' in place of a context of schema W . This enables safe instances of context weakening and context strengthening and allows us to use the reflexivity lemma (`reflG`) in the completeness proof for algorithmic equality (`ceqG`), although the function `reflG` requires a context `xaG` while the function `ceqG` provides a context `daG`. While such uses of context subsumption make proofs short and compact, and are convenient for users, one might ask whether one can avoid referring to context subsumption and make explicit the required reasoning, namely context weakening and strengthening. In other words, we eliminate context subsumption by referring to promotion lemmas such Lemma 18.

The key to express context weakening and strengthening is the ability to relate two contexts via a substitution. In Beluga^μ, we can describe context relations using *inductive datatypes* as a relation between context ϕ , context ψ and a substitution σ that maps variables from ϕ to the context γ , formally $\gamma \vdash \sigma : \phi$, as follows:

```
datatype Ctx_R : {ϕ:xaG} {γ: daG} {σ:[γ. ϕ]} ctype =
| Nil : Ctx_R [] [] [ . ^ ]
| Cons: Ctx_R [ϕ] [γ] [γ. σ]
      → Ctx_R [ϕ, b:block(x:tm,u:aeq x x)]
           [γ, b:block(x:tm,u:aeq x x,v:deq x x)] [γ,b.σ <b.1,b.2>];
```

We use first-class substitution variables σ of type $[\gamma. \phi]$ (Cave and Pientka, 2013) to provide a *witness* on how to move from the context ϕ to the context γ . We emphasize that the datatype `Ctx_R` is *not* an LF type, but instead an indexed recursive type that lives on the reasoning level. It corresponds to inductive definitions in other systems. This is indicated declaring `Ctx_R` to be a `ctype`, which is indexed by the context $\phi:xaG$, the context $\gamma:daG$ and the substitution $\sigma:[\gamma. \phi]$. In our example, the substitution σ is simply a variable substitution that weakens terms declared in the context ϕ to be meaningful in γ .

The relation `Ctx_R` is defined inductively based on the structure of contexts ϕ and γ . If ϕ and γ denote the empty context, the empty substitution, written as \wedge , acts as a witness. If σ relates ϕ and γ , then $\phi, b:block(x:tm,u:aeq x x)$ is related to $\gamma, b:block(x:tm,u:aeq x x,v:deq x x)$ by the substitution⁴ $\sigma <b.1, b.2>$ where we supply a tuple $<b.1, b.2>$ for the declaration `block (x:tm,u:aeq x x)`.

Given a term M in the term context γ and `Ctx_R [ϕ] [γ] [γ. σ]`, where σ is a weakening substitution from ϕ to γ , we can always weaken M by σ to obtain $[\gamma. M \sigma]$. This is encoded in the datatype definition below:

⁴ We take here some liberty and write simply σ ; in the theoretical foundation, substitution variables do not occur by themselves; similar to meta-variables and parameter variables they are associated with a postponed substitution.

```
datatype StrTm : (γ:daG) [γ.tm] → ctype =
| StrTm : Ctx_R [φ] [γ] [γ.σ]
  → {M: [φ.tm]} StrTm [γ. M σ];
```

We call the datatype `StrTm`, because we actually want to show that given a term `[γ.M..]` there always exists a `StrTm [γ.M..]`, i.e., there exists a term context γ , a weakening substitution σ and a term `[φ.N..]` s.t. `N σ` is equal to `M`. Here `N` is the strengthened term. This strengthening for terms is implemented by a recursive function of type:

```
str_tm : {γ:daG}{M:[γ.tm]} StrTm [γ. M.. ]
```

The proof is straightforward, but because we have defined a context relation using a recursive data-type instead of a function that when given a context ϕ of schema `xaG` returns a term context γ together with a weakening substitution σ , we need two additional lemmas⁵: we need to show that the context relation is deterministic and whenever we have a context ϕ of schema `daG`, there exists a term context γ of schema `xaG`. We can then implement promotion of reflexivity to the context `daG` as follows:

```
rec promote_refl : {γ:daG}{M:[γ.tm]} [γ.aeq (M..) (M..)] =
λ□γ ⇒ λ□M ⇒
let StrTm [γ.σ] cr [φ.N..] = str_tm [γ] [γ.M..] in
let [φ.D..] = reflG [φ] [φ.N..] in
  [γ. D σ];
```

Given a term `M` in the context γ , we first strengthen it, obtaining the strengthened term `N` in the strengthened context ϕ together with the substitution σ that maps variables in ϕ to variables in γ . We now call `reflG` with the term context ϕ it requires together with the term `N` and obtain a proof `[φ.D..]` standing for `[φ. aeq (N..) (N..)]`. Context weakening, i.e., weakening `D` s.t. it makes sense in the original context γ , is accomplished by simply applying the substitution σ .

The full development of the completeness of algorithmic equality with explicit context strengthening and promotion is available online (see `eq-proof-promotion.bel`). This development also highlights how convenient built-in support for context subsumption is in Beluga; our proof development using context subsumption (115 lines of code) is approximately a third of the proof development using explicit strengthening and promotion lemmas (318 lines of code).

4.7.2 Proofs using Context Relations

We have seen that Beluga elegantly supports proofs using generalized contexts. Here we exploit the ability to define context relations in Beluga^μ to implement proofs using context relations (R Version). To illustrate the idea, we go back to the reflexivity proof for algorithmic equality.

```
datatype Ctx_xaR : {φ:xaG} {ψ:xaG} {σ:[ψ.φ]} ctype =
| Nil_xa : Ctx_xaR [] [] [ . ^ ]
| Cons_xa : Ctx_xaR [φ] [ψ] [ψ.σ]
  → Ctx_xaR [φ, x:tm] [ψ, b:block (x:tm,u:aeq x x)] [ψ,b.σ b.1];
```

We again use first-class substitution variables σ of type `[ψ. φ]` to move from the context ϕ to the context ψ . If σ relates ϕ and ψ , then the substitution `σ b.1` relates context ϕ , `x:tm` to ψ , `b:block (x:tm,u:aeq x x)` via constructor `Cons_xa`.

⁵ Beluga^μ does not support functions in types at this point.

```

rec ctx_membership : {#p: [ $\phi$ .tm] } Ctx_xaR [ $\phi$ ] [ $\psi$ ] [ $\psi$ . $\sigma$ ]  $\rightarrow$ 
  [ $\psi$ . aeq (#p  $\sigma$ ) (#p  $\sigma$ )] =
 $\lambda^{\square}$ #p  $\Rightarrow$  fn cr  $\Rightarrow$  let (cr : Ctx_xaR [ $\phi$ ] [ $\psi$ ] [ $\psi$ . $\sigma$ ]) = cr in
case [ $\phi$ . #p..] of
| [ $\phi$ , x:tm . x]  $\Rightarrow$ 
  let Cons_xa cr' = cr in
  let (cr' : Ctx_xaR [ $\phi$ ] [ $\psi$ ] [ $\psi$ .  $\sigma$ ]) = cr' in
  [ $\psi$ , b: block (x:tm, u:aeq x x). b.2]
| [ $\phi$ , x:tm. #p..]  $\Rightarrow$ 
  let Cons_xa cr' = cr in
  let [ $\psi$ . E..] = ctx_membership [ $\phi$ .#p..] cr' in
  [ $\psi$ , b: block (x:tm,u:aeq x x) . E.. ];
rec reflR : { $\phi$ :xG}{M: [ $\phi$  . tm]} Ctx_xaR [ $\phi$ ] [ $\psi$ ] [ $\psi$  .  $\sigma$ ]  $\rightarrow$ 
  [ $\psi$  . aeq (M  $\sigma$ ) (M  $\sigma$ )] =
 $\lambda^{\square}$  $\phi$   $\Rightarrow$   $\lambda^{\square}$ M  $\Rightarrow$  fn cr  $\Rightarrow$  case [ $\phi$  . M..] of
| [ $\phi$  . #p ..]  $\Rightarrow$  ctx_membership [ $\phi$ . #p..] cr
| [ $\phi$  . app (M.. ) (N..)]  $\Rightarrow$ 
  let [ $\psi$ . D1.. ] = reflR [ $\phi$ ] [ $\phi$ .M..] cr in
  let [ $\psi$ . D2.. ] = reflR [ $\phi$ ] [ $\phi$ .N..] cr in
  [ $\psi$ . ae_a (D1..) (D2..) ]
| [ $\phi$  . lam  $\lambda$ x.M.. x]  $\Rightarrow$ 
  let [ $\psi$ , b:block (x:tm, u:aeq x x). D.. b.1 b.2] =
    reflR [ $\phi$ , x:tm] [ $\phi$ ,x:tm. M.. x] (Cons_xa cr) in
  [ $\psi$ . ae_l  $\lambda$ x. $\lambda$ u. D.. x u];

```

Fig. 9 Reflexivity of Algorithmic Equality in Beluga ^{μ} (R Version)

We show the proof `reflR` using context relations in Figure 9. It is a straightforward recursive program of type

$$\{\phi:xG\}\{M:[\phi.t\mathbf{m}]\} \text{Ctx_xaR } [\phi] [\psi] [\psi.\sigma] \rightarrow [\psi.\text{aeq } (M \sigma) (M \sigma)]$$

which can be read as “for all contexts ϕ and ψ that have schema xG and xaG respectively, if we have a substitution σ s.t. $\psi \vdash \sigma : \phi$, then for all terms M depending on ϕ , we have a proof that $[\psi.\text{aeq } (M \sigma) (M \sigma)]$. We note that because the term M depends only on the context ϕ , we explicitly weaken it by applying σ to move it to the context ψ .

The proof is implemented as a recursive function over the term M . We again consider three possible cases: If we have an application $[\phi.\text{app } (M1..)(M2..)]$, we appeal to the induction hypothesis on $[\phi.M1]$ and $[\phi.M2..]$ by making a recursive call. Since the context ϕ and the context ψ does not change, we can simply make the recursive all on $[\phi.M1..]$ and $[\phi.M2..]$ respectively using the relation `cr`.

When we have $[\phi.\text{lam } \lambda x.M..x]$, we want to appeal to the induction hypothesis on $[\phi.x:tm. M..x]$. To make the recursive call, we also need a witness relating the context $[\phi.x:tm. M..x]$ to the context $[\psi,b:block (x:tm,u:aeq x x)]$. Recall that `cr` stands for `Ctx_xaR [ϕ] [ψ] [ψ . σ]`. Therefore, by `Cons_xa`, we know there exists `Ctx_xaR [ϕ ,x:tm] [ψ ,b:block (x:tm,u:aeq x x)] [ψ ,b. σ b.1]` and we appeal to the induction hypothesis by `reflR [ϕ ,x:tm] [ϕ ,x:tm.M..x] (Cons_xa cr)`.

Finally, we take a close look at the variable case. For clarity, we follow the outline of the proof from the companion paper (Felty et al, 2014) and factor out the Context Membership Lemma 6 (see Sec. 2.2). We distinguish two different cases depending on the position of the variable in the context by pattern matching on

the shape of ϕ . If $[\phi, x:\text{tm}.x]$, then we inspect the context relation cr . We note that pattern matching forces the original context ϕ to be $\phi, x:\text{tm}$. By pattern matching on cr' , we observe that there exists a relation cr' , s.t. $\text{Ctx_xaR } [\phi] [\psi] [\psi.\sigma]$. Moreover, $\psi = \psi, b:\text{block } (x:\text{tm}, u:\text{aeq } x \ x)$ and $\sigma = \sigma \text{ b.1}$ where the left hand side denotes the original context and substitution, while the right hand side shows the context and substitution refinement after pattern matching. We must show that there exists a proof for $\text{aeq } x \ x$ in the context $\psi, b:\text{block } (x:\text{tm}, u:\text{aeq } x \ x)$. This is simply b.2 .

If instead $[\phi, x:\text{tm}.\#p \dots]$, we have a variable other than x . In this case, we also observe that the context $\phi, x:\text{tm}$ and $\psi, b:\text{block } (x:\text{tm}, u:\text{aeq } x \ x)$ are related by the substitution $\sigma \text{ b.1}$ since we have the assumption cr . By inversion on Cons_xa we know that cr' relates ϕ and ψ by the substitution σ . We now appeal to the induction hypothesis making the recursive call $\text{ctx_membership } [\phi.\#p \dots] \text{ cr}'$. This is a valid recursive call, since the context ϕ is smaller than the original context. We then rely on declaration weakening to obtain the desired result⁶.

Substitution variables account, in principle, for a concise representation of proofs using context relations; they are particularly useful to express elegantly proofs by logical relations (Cave and Pientka, 2014) where substitutions play an essential role. In the completeness proof of algorithmic equality, they make possible the implementation of the proof using context relations. However, the overhead of defining context and equivalence relations and proving that the defined relations are total functions is substantial in the R version of the completeness proof of algorithmic equality⁷.

5 The Two-Level Approach

In this section we briefly review the two-level architecture that Abella and Hybrid share, referring to (Felty and Momigliano, 2012) and (Gacek et al, 2012) for more extensive explanation.

In this approach we distinguish between a simple *specification logic* (SL), where we specify (and execute) our judgments, and a *reasoning* logic (RL), endowed with some form of induction/recursion, where we conduct our proofs. Abella and Hybrid use different RLs, while they basically share the same SL, namely a fragment of *second-order* hereditary Harrop formulas.⁸ The SL language is defined as follows, where A is an atomic formula. We use boldface font writing \mathbf{x} to mean a vector of variables x_1, \dots, x_n in $\forall \mathbf{x} (G \supset A)$. We extend our vector notation to substitutions writing $[\mathbf{t}/\mathbf{x}]G$ as a shorthand for $[t_1/x_1, \dots, t_n/x_n]G$.

$$\begin{aligned} \text{Clauses } C &::= \forall \mathbf{x} (G \supset A) \\ \text{Goals } G &::= \top \mid A \mid G_1 \wedge G_2 \mid A \supset G \mid \forall x. G \mid \exists x. G \end{aligned}$$

⁶ Type checking this proof requires the experimental implementation of Beluga^h supporting substitution variables available at <http://www.cs.mcgill.ca/~complogic/beluga/>.

⁷ The completeness proof of algorithmic equality (G version) in Beluga is 115 lines of code compared to its R version which takes 427 lines of code.

⁸ The two-level architecture makes possible, indeed it encourages, to plug-in a different SL; in fact, Abella 2.0 (Wang et al, 2013) features an n-ary version of Harrop formulas, whereby *focusing* generalises backchaining. Because all the benchmarks that we study here are second-order only, we take the liberty of describing here this version of the SL as the specialization of the n-ary one to our examples. Note however that the scripts described in the Abella section run under Abella 2.0 and that some of the relative differences are explained thereby.

$$\begin{array}{c}
\frac{}{\Sigma; \Gamma \rightarrow_{\mathcal{C}} \top} \top_R \quad \frac{\Sigma; \Gamma \rightarrow_{\mathcal{C}} G_1 \quad \Sigma; \Gamma \rightarrow_{\mathcal{C}} G_2}{\Sigma; \Gamma \rightarrow_{\mathcal{C}} G_1 \wedge G_2} \wedge_R \\
\frac{(\Sigma, a); \Gamma \rightarrow_{\mathcal{C}} G[a/x]}{\Sigma; \Gamma \rightarrow_{\mathcal{C}} \forall x. G} \forall_R \quad \frac{\Sigma; \Gamma \rightarrow_{\mathcal{C}} G[t/x]}{\Sigma; \Gamma \rightarrow_{\mathcal{C}} \exists x. G} \exists_R \quad \frac{\Sigma; (\Gamma, A) \rightarrow_{\mathcal{C}} G}{\Sigma; \Gamma \rightarrow_{\mathcal{C}} A \supset G} \supset_R \\
\frac{}{\Sigma; (\Gamma, A) \rightarrow_{\mathcal{C}} A} \text{init} \quad \frac{\Sigma; \Gamma \rightarrow_{\mathcal{C}} [t/x]G}{\Sigma; \Gamma \rightarrow_{\mathcal{C}} A} \text{bc}
\end{array}$$

The rule bc has the proviso that $\forall \mathbf{x} (G \supset A') \in \mathcal{C}, [t/\mathbf{x}]A' = A$.

Fig. 10 A Minimal Sequent Calculus with Backchaining

Note that this SL is less powerful than the logical framework LF, the specification language of Twelf and Beluga as it restricts implications to $A \supset G$, i.e., we only make atomic assumptions. This language is simple enough that its sequent calculus is analogous to a logic programming interpreter. In fact, we can write inference rules so that the only left rule is similar to Prolog's backchaining. Sequents have the form $\Sigma; \Gamma \rightarrow_{\mathcal{C}} G$, where Σ is the current signature of eigenvariables and we distinguish clauses belonging to a static database \mathcal{C} , intuitively the specification, from formulas in Γ , the atoms introduced by the right implication rule. The rules for this logic are given in Figure 10.

Note that in the proof-theoretic setting, as opposed to the type-theoretic one, a declaration block in a context such as $(\mathbf{x} : \mathbf{tm}; \mathbf{aeq} \ \mathbf{x} \ \mathbf{x})$ is split into two parts: a *generic* part, which essentially only records that \mathbf{x} is indeed a (fresh) eigenvariable of sort \mathbf{tm} and a *hypothetical* one, which states the relevant assumption for the given eigenvariable. Hence the separation between Σ and Γ , although Hybrid and Abella will differ on how genericity is realized. Genericity, in fact, is a property of the quantification structure of the RL, while hypothetical judgments are made possible by encapsulation inside the SL.

The SL and object level judgments are themselves encoded in the RL as *inductive* definitions with the following types:

```

seq  : atm list → nat → o → prop.
prog : atm → o → prop.

```

where **prop** is the kind of propositions in the RL, **atm** represents atomic predicates of the object-language, and **o** is the type of SL propositions. Natural numbers (**nat**) decorate sequents, in order to reason by (complete) induction on the *height* of a proof. The logic connectives have sorts

$$\begin{array}{l}
\top : \mathbf{o} \mid \langle _ \rangle : \mathbf{atm} \rightarrow \mathbf{o} \mid \wedge : \mathbf{o} \rightarrow \mathbf{o} \rightarrow \mathbf{o} \mid \supset : \mathbf{atm} \rightarrow \mathbf{o} \rightarrow \mathbf{o} \mid \\
\forall_{\tau} : (\tau \rightarrow \mathbf{o}) \rightarrow \mathbf{o} \mid \exists_{\tau} : (\tau \rightarrow \mathbf{o}) \rightarrow \mathbf{o}.
\end{array}$$

where $\langle _ \rangle$ coerces atoms into SL propositions.⁹ In the definition of the SL (see Fig. 11), we write $\{\Gamma \vdash_n G\}$ to pretty-print the **seq** definition and we overload the usual logical symbols. Free variables in inductive definitions and also in statements of theorems are implicitly universally quantified at the top-level of each clause or

⁹ The τ argument to quantifiers ranges in Abella over any sort, while in Hybrid it is restricted to **expr**, the built-in sort of object-level expressions. We discuss this issue further in Section 8.

$$\begin{array}{l}
\mathbf{s.t} : \\
\mathbf{s.and} : \quad \{\Gamma \vdash_n G_1\} \rightarrow \{\Gamma \vdash_n G_2\} \rightarrow \{\Gamma \vdash_{n+1} \top\} \\
\mathbf{s.all} : \quad \mathbf{Q}x. \{\Gamma \vdash_n G x\} \rightarrow \{\Gamma \vdash_{n+1} \forall_r(\lambda x. G x)\} \\
\mathbf{s.ex} : \quad \{\Gamma \vdash_n G M\} \rightarrow \{\Gamma \vdash_{n+1} \exists_r(\lambda x. G x)\} \\
\mathbf{s.imp} : \quad \{A :: \Gamma \vdash_n G\} \rightarrow \{\Gamma \vdash_{n+1} A \supset G\} \\
\mathbf{s.init} : \quad A \in \Gamma \rightarrow \{\Gamma \vdash_{n+1} \langle A \rangle\} \\
\mathbf{s.bc} : \quad \mathbf{prog} A G \rightarrow \{\Gamma \vdash_n G\} \rightarrow \{\Gamma \vdash_{n+1} \langle A \rangle\}
\end{array}$$

Fig. 11 Encoding of Specification Logic

statement. Contexts are represented as lists,¹⁰ the `::` operator represents cons, and the `∈` operator list membership.

The first five clauses of the definition directly encode the introduction rules of a sequent calculus for this logic. The rule `s_all` maps the SL universal quantifier to the related universal in the RL (here ambiguously denoted by \mathbf{Q}), which in the case of Hybrid will be Coq’s dependent product and in the case of Abella the ∇ quantifier. In the subsequent discussion, we prefix theorems with A (resp. H) writing A -Theorem (resp. H -Theorem) if we are referring to Abella’s or Hybrid’s encoding. In `s_ex` we rely on meta-logic unification to instantiate the logic variable M .¹¹ In the last two rules, atoms are provable either by assumption or via *backchaining* over Prolog-like rules, the `prog` predicate, which encode the properties of the object-language. The sequent calculus is parametric in those clauses and uses the RL underlying unification to do the matching required by the `s_bc` rule.

For example, clause `ae_1` in Appendix A is encoded as the following `prog` clause, where we do *not* explicitly encode the outermost SL universal quantifier, but we short-circuit it with the one of the meta-logic, hence omitted:

```

prog (aeq (lam (λx. M x)) (lam (λx. N x)))
    (∀ (λx. aeq x x ⊃ <aeq (M x) (N x)>)).

```

In the rest of this section we will use the more readable “prog-less” syntax of ORBI. As explained in the companion paper, ORBI specifications need to be massaged to fit the proof-theoretic view at the level of inference rules, schemas, and context relations. This is mostly due to the asymmetry between declarations that give typing (sorting) information, e.g., `tm:type`, and those expressing judgments, e.g., `aeq:tm -> tm -> type`. The logic typically does not know that $M:\mathbf{tm}$ entails `is_tm M` holding at the level of judgments, because the language of *sorts* tends not to be reflected in the proof-theory. Therefore, for the type `tm`, we introduce the predicate `is_tm` with the following clauses:

```

tm_a: is_tm M → is_tm N → is_tm (app M N).
tm_l: ({x} is_tm x → is_tm (M x)) → is_tm (lam (λx. M x)).

```

These well-formedness rules have two important applications:

1. to provide induction principles at the level of the syntax as for example required in the proof of admissibility of reflexivity (Theorem 7); see e.g. H -Theorem 2 and A -Theorem 2 ;

¹⁰ Using lists as the data-structure underlying contexts is not written in stone. In fact, the Isabelle/HOL version of Hybrid uses sets (Felty and Momigliano, 2012) and Abella may also offer multi-sets in the near future.

¹¹ Abella does not support existentials goals, and, to overcome this, the user needs to resort to some indirect encoding, as exemplified in (Momigliano, 2012).

2. to prove preservation of well-formedness results such as H-Theorem 30 and A-Theorem 9.¹²

In fact, with respect to the latter, we assumed in our presentation in the companion paper that whenever a judgment was provable, the terms in it were well-formed. In the two-level approach, such statements need to be proved explicitly, and to do so, annotations are added to inference rule encodings. In Hybrid and Abella, for instance, clauses `de_l` and `de_r` will have the form :

```
de_l: ({x} is_tm x → deq x x → deq (M x) (N x))
      → deq (lam (λx. M x)) (lam (λx. N x)).
de_r: is_tm M → deq M M.
```

In order to avoid clutter, annotations that are not needed for adequacy are left out; this is the case for the `ae_l` rule above. We take the same approach when specifying context relations.

A small set of structural rules of the SL is proved once and for all, and then used to reason about object-languages:

SL-Theorem 1 (Structural Properties)

- (a) Height weakening: $\{\Gamma \vdash_n G\} \rightarrow n < m \rightarrow \{\Gamma \vdash_m G\}$.
- (b) Antecedent weakening: $\{\Gamma \vdash_n G\} \rightarrow \Gamma \subseteq \Gamma' \rightarrow \{\Gamma' \vdash_n G\}$.
- (c) Atomic cut: $\{A :: \Gamma \vdash_n G\} \rightarrow \{\Gamma \vdash_m \langle A \rangle\} \rightarrow \{\Gamma \vdash_{n+m} G\}$.
- (d) Specialization: $\{\Gamma \vdash_n \forall x. G x\} \rightarrow \{\Gamma \vdash_n G t\}$.

Weakening is defined “set-theoretically”, i.e. $\Gamma \subseteq \Gamma'$ iff $\forall A. A \in \Gamma \rightarrow A \in \Gamma'$, and, as such, it subsumes contraction and in particular:

$$\text{Exchange: } \Gamma \subseteq \Gamma' \wedge \Gamma' \subseteq \Gamma \rightarrow \{\Gamma \vdash_n G\} \text{ iff } \{\Gamma' \vdash_n G\}.$$

Declaration strengthening *d-str* is currently unsupported, and thus it is established on a case-by-case basis, typically with inductive proofs, although it is conceivable that it could be proven once and for all by enriching the `prog` clause with a dependency analysis, akin to Twelf and Beluga’s subordination. Context strengthening *c-str* also requires separate proofs and its automation seems more complex.

As a consequence of the proof-theoretic separation between generic and hypothetical judgments, the *substitution* property is realized by a combination of cut and specialization. See for example the proof of A-Theorem 19, viz. the `pr_b` case. Specialization is just function application in Coq, while it is a rather deep property of the nabla quantifier and SL-derivability.

Finally, context schemas/relations are realized by *inductive definitions* of certain predicates/relations in the RL. For example, schema `daG` (see Appendix A.1) corresponds to a unary predicate `daG: atm list → prop` inductively defined as:

```
daG nil ;
Q x. daG L → daG (tm x :: deq x x :: aeq x x :: L).
```

The block structure is thus enforced by pattern matching on the `cons` notation, which in this case requires the list to either be empty or to always contain the declaration `deq x x` followed by `aeq x x` for some `Q`-quantified variable `x`.

¹² We refer to these properties as “internal adequacy”, as opposed to “external adequacy”; the latter refers to the bijection that should hold between the mathematical presentation of a OL and its encoding in a logical framework. See the discussion in (Pfenning, 2001), (Gacek et al, 2012) and (Felty and Momigliano, 2012) for Twelf, Abella and Hybrid respectively.

6 Mechanization in Hybrid

In the version of Hybrid used in this paper, the RL is Coq’s Calculus of Inductive Constructions. We use a pretty-printed version of Coq concrete syntax, which includes some notation already presented. For example, **Prop** is the type of propositions of the RL, corresponding to **prop** in the previous section. The type **o** from the previous section is defined here as the following Coq inductive type:

$$\text{Inductive } \mathbf{o} := \text{tt} : \mathbf{o} \mid \langle _ \rangle : \text{atm} \rightarrow \mathbf{o} \mid _ \text{and} _ : \mathbf{o} \rightarrow \mathbf{o} \rightarrow \mathbf{o} \mid _ \text{imp} _ : \text{atm} \rightarrow \mathbf{o} \rightarrow \mathbf{o} \\ \mid \text{all} : (\text{expr} \rightarrow \mathbf{o}) \rightarrow \mathbf{o} \mid \text{ex} : (\text{expr} \rightarrow \mathbf{o}) \rightarrow \mathbf{o}.$$

where, although we overloaded the logical connectives of the RL and SL in Section 5, this definition introduces new symbols for the SL connectives so that we may more clearly distinguish them in this section. Hybrid is untyped in the sense that all object-level data have type **expr**. It provides a set of operators on this type that is built definitionally on the foundation of the meta-language of the underlying theorem prover using a form of de Bruijn indices to represent binding operators; no axioms are introduced (see Felty and Momigliano, 2012). We omit all details and just present the operators that are used in this paper, with their types, which are:

$$\begin{array}{ll} \text{CON} : \text{con} \rightarrow \text{expr} & \text{APP} : \text{expr} \rightarrow \text{expr} \rightarrow \text{expr} \\ \text{VAR} : \text{var} \rightarrow \text{expr} & \text{LAM} : (\text{expr} \rightarrow \text{expr}) \rightarrow \text{expr} \end{array}$$

where the first three come directly from the inductive definition of **expr**, and **LAM** is defined and provides the capability to express OL syntax using HOAS. Also, **var** is defined to be the natural numbers and can be used to represent free variables, and **con** will be defined later to represent the constants of an OL. These constants, together with **APP** and **LAM**, will be used to define operators for OL syntax (such as **app** and **lam** of the untyped lambda-calculus). Once defined, such definitions never need to be expanded by the user.¹³

Figure 11 is mapped fairly directly to a Coq inductive definition of type **list atm** \rightarrow **nat** \rightarrow **o** \rightarrow **Prop**, in most cases by simply replacing the connectives shown in the figure with those in the definition of type **o**. The only clause that differs is the one for generic judgments, which in Hybrid is:

$$\text{s_all} : (\forall x. \text{proper } x \rightarrow \{ \Gamma \vdash_n G \ x \}) \rightarrow \{ \Gamma \vdash_{n+1} (\text{all } G) \}.$$

Note the use of the universal quantifier of the RL. By using the quantifier, we do not distinguish variables occurring in a context from terms. The consequences of this choice will be discussed later. The use of the **proper** annotation rules out terms that have occurrences of bound variables that do not have a corresponding binder (*dangling indices*).¹⁴ Explicit proof heights are included in sequents and many proofs will be by complete induction on the height of a proof. In this section

¹³ Expanding them fully down to primitives gives the low-level de Bruijn representation, which is hidden from the user when reasoning about meta-theory.

¹⁴ Hybrid 0.2 described in Momigliano et al (2008), which is implemented in Isabelle/HOL, includes an improvement that doesn’t require the **proper** predicate, but this improvement has not yet been ported to the Coq version used in this paper.

```

Inductive prog : atm → o → Prop :=
tm_a :          prog (is_tm (app M N)) ((is_tm M) and (is_tm N))
tm_l :          abstr M → prog (is_tm (lam M))
                all λx. (is_tm x) imp (is_tm (Mx))
ae_a :          prog (aeq (app M1 M2) (app N1 N2))
                (aeq M1 N1) and (aeq M2 N2)
ae_l : abstr M → abstr N → prog (aeq (lam M) (lam N))
                all λx. (aeq x x) imp (aeq (Mx) (Nx)).

```

Fig. 12 Object-Language Inference Rules in Hybrid

$\{ \Gamma \vdash G \}$ is an abbreviation for $\exists n. \{ \Gamma \vdash_n G \}$. We distinguish sequents of the SL from those of the RL by writing the latter as:

$$\frac{\mathbf{h}_1 : A_1 \quad \mathbf{h}_2 : A_2 \quad \cdots \quad \mathbf{h}_n : A_n}{A}$$

where A_1, \dots, A_n, A are Coq formulas (of type `Prop`) and $\mathbf{h}_1, \dots, \mathbf{h}_n$ are hypothesis names. (The latter are sometimes omitted.) Such sequents appear in the description of formal proofs, which are often described in a goal-directed manner. When there is more than one subgoal with the same set of hypotheses, we write more than one conclusion below the line and number them.

To illustrate the Coq encoding of OL inference rules, we present the syntax of the untyped lambda-calculus and `prog` clauses for algorithmic equality (`ae_e` and `ae_l`) as presented in Section 2.1. We fill in the definition of `con`, define operators `app` and `lam` using the operators defined earlier for `expr`, and fill in the definition of `atm`, which includes `aeq` as well as the `is_tm` predicate for well-formedness of terms.

```

Inductive con  := cAPP : con | cLAM : con.
Definition app := λM, N. APP (APP (CON cAPP) M) N.
Definition lam := λM. APP (CON cLAM) (LAM M).
Inductive atm := is_tm : expr → atm | aeq : expr → expr → atm.

```

The inference rules are defined as the inductive definition of `prog` in Figure 12. Both the definition of `prog` and the definitions above are automatically generated by the tool described in ?, which translates ORBI files to Coq script. This script imports the Hybrid libraries and sets up the basic definitions needed to reason about a particular OL, including definitions for context schemas and relations.

Two reasons for including the `prog` clauses `tm_l` and `tm_a` were discussed in the previous section. First, they provide induction on well-formedness derivations, which is often useful, especially since induction directly on terms is not currently available.¹⁵ Second, they are important for proving internal adequacy. The latter is especially important in Hybrid, since as mentioned above, Hybrid expressions are untyped. In Hybrid, we also include `abstr` conditions, which rule out meta-level functions that do not encode object-level syntax. An example is discussed in Section 6.3.3; more details on both external and internal adequacy can be found in Felty and Momigliano (2012).

¹⁵ Please see Ambler et al (2003) and Capretta and Felty (2007) for work in this direction.

The `prog` clauses for all other judgments considered here are encoded via translation from the specifications in the `Rules` sections of the ORBI files (see Appendix A) in a systematic manner. The only non-systematic aspect of the encoding of inference rules is the insertion of well-formedness annotations. (The same ones are used in both Hybrid and Abella.) We add special directives to the ORBI files (which are not shown in the appendix, but are discussed in the companion paper) that indicate which annotations to include.

For all benchmark problems that we are able to formalize in Hybrid, we do both the R and G versions, sometimes relegating one of the two versions to the appendix.

6.1 Basic Linear Context Extension

6.1.1 R Version

We begin by defining the context relation `xaR` (see the `Definitions` section of Appendix A.1) in Coq:

```
Inductive xaR : list atm → list atm → Prop :=
  xaR nil nil
  xaR  $\Phi_x$   $\Phi_a$  → proper  $x$  → xaR (is_tm  $x$  ::  $\Phi_x$ ) (aeq  $x$   $x$  ::  $\Phi_a$ ).
```

Recall our convention of leaving out explicit quantifiers at the top-level, which here includes universal quantification over x as well as Φ_x and Φ_a in the second clause of the definition. Here x is an arbitrary term, since as mentioned, we do not distinguish variables occurring in a context from terms. Note that whenever the typing declaration `x:tm` appears in a context in an ORBI specification, the corresponding Hybrid definition contains the atomic formula `is_tm x`. (Similarly, in the next subsection, declarations of the form `a:tp` in ORBI will appear as formulas in Hybrid.) Also note that `aeq x x` appears by itself instead of in a block with `is_tm x`. As stated in the previous section, we leave out well-formedness annotations from `prog` clauses and context relations whenever possible. Finally, note also the presence of the required `proper` annotation.

We start with the Hybrid version of membership Lemma 6 from Section 2.2.

H-Lemma 1 (Context Membership)

$$\forall M, \Phi_x, \Phi_a. \text{xaR } \Phi_x \Phi_a \rightarrow \text{is_tm } M \in \Phi_x \rightarrow \text{aeq } M M \in \Phi_a.$$

The proof is by induction on the first premise, and is simple and follows the same pattern as all other context membership lemmas. Current work on Hybrid includes writing a Coq tactic to automate these proofs, which will be straightforward. Note that the second implication in the statement of this lemma could be replaced by equivalence, since it certainly holds in both directions. We only state such lemmas using implication in the direction(s) that are needed, so that they can be used in automating proofs via `eauto`. The `eauto` tactic is parametrized by a set of lemmas marked as hints and we generally add context membership, strengthening, and weakening lemmas as hints, which together provide significant automation of proofs.

We give the proof of the first benchmark in some detail.

H-Theorem 2 (Admissibility of Reflexivity, R Version)

$$\mathbf{xaR} \Phi_x \Phi_a \rightarrow \{\Phi_x \vdash_n \langle \mathbf{is_tm} M \rangle\} \rightarrow \{\Phi_a \vdash_n \langle \mathbf{aeq} M M \rangle\}.$$

Proof The proof is by complete induction on n with induction hypothesis $IH := \forall i, M, \Phi_x, \Phi_a. i < n \rightarrow \mathbf{xaR} \Phi_x \Phi_a \rightarrow \{\Phi_x \vdash_i \langle \mathbf{is_tm} M \rangle\} \rightarrow \{\Phi_a \vdash_i \langle \mathbf{aeq} M M \rangle\}$.

A derivation of $\{\Phi_x \vdash_n \langle \mathbf{is_tm} M \rangle\}$ must end in an application of the last two clauses of the definition of the SL ($\mathbf{s_init}$ or $\mathbf{s_bc}$ in Figure 11 on page 27). In general, the $\mathbf{s_init}$ case in Hybrid proofs corresponds to the variable case in the informal proof. Here, we know that $(\mathbf{is_tm} M) \in \Phi_x$. By H-Lemma 1, $(\mathbf{aeq} M M) \in \Phi_a$. We use this fact and simply apply $\mathbf{s_init}$ to obtain $\{\Phi_a \vdash_n \langle \mathbf{aeq} M M \rangle\}$, as desired.

All other cases in the informal proofs are covered by the $\mathbf{s_bc}$ case in Hybrid proofs. Here, when the derivation ends in $\mathbf{s_bc}$, it must be the case that one of the two clauses defining the $\mathbf{is_tm}$ judgment ($\mathbf{tm_a}$ or $\mathbf{tm_l}$ in Figure 12 on page 30) was used.

Case $\mathbf{tm_a}$: We know that M has the form $(\mathbf{app} M_1 M_2)$, and we must show:

$$\frac{\mathbf{h}_1 : IH \quad \mathbf{h}_2 : \mathbf{xaR} \Phi_x \Phi_a \quad \mathbf{h}_3 : \{\Phi_x \vdash_n \langle \mathbf{is_tm} (\mathbf{app} M_1 M_2) \rangle\}}{\{\Phi_a \vdash_n \langle \mathbf{aeq} (\mathbf{app} M_1 M_2) (\mathbf{app} M_1 M_2) \rangle\}}$$

By repeated inversion of the SL rules on \mathbf{h}_3 , and repeated backward application of these rules to the conclusion, the above goal reduces to the following 2 subgoals:

$$\frac{IH \quad \mathbf{xaR} \Phi_x \Phi_a \quad \{\Phi_x \vdash_{n-2} \langle \mathbf{is_tm} M_1 \rangle\} \quad \{\Phi_x \vdash_{n-2} \langle \mathbf{is_tm} M_2 \rangle\}}{1. \{\Phi_a \vdash_{n-2} \langle \mathbf{aeq} M_1 M_1 \rangle\} \\ 2. \{\Phi_a \vdash_{n-2} \langle \mathbf{aeq} M_2 M_2 \rangle\}}$$

which are both provable by the induction hypothesis.

Case $\mathbf{tm_l}$: We know that M has the form $(\mathbf{lam} M')$ and we must show:

$$\frac{\mathbf{h}_1 : IH \quad \mathbf{h}_2 : \mathbf{xaR} \Phi_x \Phi_a \quad \mathbf{h}_3 : \{\Phi_x \vdash_n \langle \mathbf{is_tm} (\mathbf{lam} M') \rangle\}}{\{\Phi_a \vdash_n \langle \mathbf{aeq} (\mathbf{lam} M') (\mathbf{lam} M') \rangle\}}$$

Again, by repeated inversion of the SL rules on \mathbf{h}_3 , and repeated backward application of these rules to the conclusion, the above goal reduces to the following subgoal:

$$\frac{IH \quad \mathbf{xaR} \Phi_x \Phi_a \quad \mathbf{proper} x \quad \{\mathbf{is_tm} x :: \Phi_x \vdash_{n-3} \langle \mathbf{is_tm} (M' x) \rangle\}}{\{\mathbf{aeq} x x :: \Phi_a \vdash_{n-3} \langle \mathbf{aeq} (M' x) (M' x) \rangle\}}$$

By definition of \mathbf{xaR} , we can conclude $\mathbf{xaR} (\mathbf{is_tm} x :: \Phi_x) (\mathbf{aeq} x x :: \Phi_a)$, and then apply the induction hypothesis to complete the proof.

6.1.2 G Version

We begin by defining the generalized context inductively.¹⁶

$$\begin{aligned} \mathbf{Inductive} \ \mathbf{xaG} : \mathbf{list} \ \mathbf{atm} \rightarrow \mathbf{Prop} := \quad & \mathbf{xaG} \ \mathbf{nil} \\ & \mathbf{xaG} \ \Phi_{x_a} \rightarrow \mathbf{proper} \ x \rightarrow \mathbf{xaG} \ (\mathbf{is_tm} \ x :: \mathbf{aeq} \ x \ x :: \Phi_{x_a}). \end{aligned}$$

¹⁶ We omit the Coq inductive definitions for contexts in the rest of this section. G versions like this one are obtained directly from the corresponding definition in the `Schemas` section of the ORBI files in the appendix. Similarly, inductive definitions of context relations such as \mathbf{xaR} are obtained from the corresponding `Definitions` sections.

In order to formalize the lambda case of Theorem 8 we need a *d-str* lemma. Here, we state it as a corollary (H-Corollary 4 below) of the more general H-Lemma 3. (Although strengthening was not needed in the informal proof, it is required here because of the fact that we leave out assumptions of the form `is_tm x` when defining the inference rules for `aeq`. See Figure 12 on page 30).

H-Lemma 3 $(\forall x, y. \text{aeq } x \ y \in \Gamma_1 \iff \text{aeq } x \ y \in \Gamma_2) \rightarrow \{\Gamma_1 \vdash_n \langle \text{aeq } M \ N \rangle\} \rightarrow \{\Gamma_2 \vdash_n \langle \text{aeq } M \ N \rangle\}$.

Note that this lemma preserves the height n . It is proved by a straightforward induction on proof height n . We state it in this general form because other instances of both *d-str* and *c-str* required later also follow directly from it.

H-Corollary 4 (D-Strengthening)

$\{\text{is_tm } x :: \text{aeq } x \ x :: \Gamma \vdash_n \langle \text{aeq } M \ N \rangle\} \rightarrow \{\text{aeq } x \ x :: \Gamma \vdash_n \langle \text{aeq } M \ N \rangle\}$.

We also need the standard membership lemma (omitted). With these in place, the proof of the admissibility of reflexivity is straightforward. We show it in some detail for comparison.

H-Theorem 5 (Admissibility of Reflexivity, G Version)

$\text{xaG } \Phi_{xa} \rightarrow \{\Phi_{xa} \vdash_n \langle \text{is_tm } M \rangle\} \rightarrow \{\Phi_{xa} \vdash_n \langle \text{aeq } M \ M \rangle\}$.

Proof Again, the proof is by complete induction on n . The context and application cases are similar to H-Theorem 2, just like these two cases were similar in the informal proofs of Theorems 7 and 8 presented in the companion paper (Felty et al, 2014). In the abstraction case (`tm_1`), we know that M has the form $(\text{lam } M')$ and we must show:

$$\frac{\text{h}_1 : IH \quad \text{h}_2 : \text{xaG } \Phi_{xa} \quad \text{h}_3 : \{\Phi_{xa} \vdash_n \langle \text{is_tm } (\text{lam } M') \rangle\}}{\{\Phi_{xa} \vdash_n \langle \text{aeq } (\text{lam } M') \ (\text{lam } M') \rangle\}}$$

By repeated inversion of the SL rules on h_3 , and repeated backward application of these rules to the conclusion, the above goal reduces to the following subgoal:

$$\frac{IH \quad \text{xaG } \Phi_{xa} \quad \text{h}_4 : \text{proper } x \quad \text{h}_5 : \{\text{is_tm } x :: \Phi_{xa} \vdash_{n-3} \langle \text{is_tm } (M'x) \rangle\}}{\{\text{aeq } x \ x :: \Phi_{xa} \vdash_{n-3} \langle \text{aeq } (M'x) \ (M'x) \rangle\}}$$

We apply *d-wk* to h_5 via SL-Theorem 1 (b) to obtain:

$$\{\text{is_tm } x :: \text{aeq } x \ x :: \Phi_{xa} \vdash_{n-3} \langle \text{is_tm } (M'x) \rangle\}$$

By definition of `xaG`, we can conclude `xaG (is_tm x :: aeq x x :: Φxa)`, and then apply *IH* to obtain $\{\text{is_tm } x :: \text{aeq } x \ x :: \Phi_{xa} \vdash_{n-3} \langle \text{aeq } (M'x) \ (M'x) \rangle\}$. Finally, we apply *d-str* via H-Corollary 4 to obtain $\{\text{aeq } x \ x :: \Phi_{xa} \vdash_{n-3} \langle \text{aeq } (M'x) \ (M'x) \rangle\}$ as desired.

For admissibility of reflexivity, which is our simplest benchmark, there is very little difference between the R and G versions in Hybrid. As we get to more complicated examples, the G versions require more (sometimes many more) strengthening, weakening, and promotion lemmas as compared to the R versions, but other than that, present no further complications.

The proofs of admissibility of symmetry and transitivity for algorithmic equality use the following membership lemma.

H-Lemma 6 (Context Inversion) $\mathbf{aG} \Phi_a \rightarrow \mathbf{aeq} M N \in \Phi_a \rightarrow M = N$.

Here the inductive definition of \mathbf{aG} is obtained from the definition of schema \mathbf{xaG} in Appendix A.1, in this case omitting the well-formed term annotations, i.e., in $(\mathbf{aG} \Phi_a)$, the context Φ_a contains only atoms of the form $(\mathbf{aeq} x x)$. Like reflexivity, the proofs of symmetry and transitivity are straightforward and are stated as follows in Hybrid.

H-Theorem 7 (Symmetry and Transitivity)

1. $\mathbf{aG} \Phi_a \rightarrow \{\Phi_a \vdash_n \langle \mathbf{aeq} M N \rangle\} \rightarrow \{\Phi_a \vdash_n \langle \mathbf{aeq} N M \rangle\}$.
2. $\mathbf{aG} \Phi_a \rightarrow \{\Phi_a \vdash_n \langle \mathbf{aeq} M L \rangle\} \rightarrow \{\Phi_a \vdash_n \langle \mathbf{aeq} L N \rangle\} \rightarrow \{\Phi_a \vdash_n \langle \mathbf{aeq} M N \rangle\}$.

We note here that the abstraction cases of some formal proofs require explicit reasoning about Hybrid's extensional equality (Felty and Momigliano, 2012). The only two examples in this paper where such details appear are the \mathbf{lam} cases of H-Theorem 7 (2) above and the failed proof of H-Attempt 20 in Appendix B.2.

6.2 Linear Context Extensions with Alternative Declarations

The next benchmark extends reflexivity of equality to the polymorphic lambda-calculus. We first extend the definitions on page 30 to include new constants and operators \mathbf{tapp} and \mathbf{tlam} and replace the definition of \mathbf{atm} with one that includes four predicates: $\mathbf{is_tp}$, $\mathbf{is_tm}$, \mathbf{atp} , and \mathbf{aeq} , thus implementing the **Syntax** and **Judgments** sections of the ORBI file in Appendix A.2. The definition of \mathbf{prog} in Figure 12 is extended to include all the declarations in the **Rules** section, plus clauses for well-formedness of \mathbf{tp} .

6.2.1 G Version

The inductive definitions for context schemas \mathbf{atpG} and \mathbf{aeqG} implement directly the corresponding schema declarations in Appendix A.2, where the former includes blocks of the form $(\mathbf{is_tp} \alpha :: \mathbf{atp} \alpha \alpha)$ and the latter includes, in addition, blocks of the form $(\mathbf{is_tm} x :: \mathbf{aeq} x x)$. The statement and proof of admissibility of reflexivity for types is similar to the case for terms in the untyped lambda-calculus. We simply state the required lemmas and theorem. The lemmas include the usual membership lemma as well as a *d-str* lemma and corollary (similar to H-Lemma 3 and H-Corollary 4).

H-Lemma 8 (Context Membership)

$$\mathbf{atpG} \Phi_{atp} \rightarrow \mathbf{is_tp} A \in \Phi_{atp} \rightarrow \mathbf{atp} A A \in \Phi_{atp}.$$

H-Lemma 9 $(\forall \alpha, \alpha'. \mathbf{atp} \alpha \alpha' \in \Gamma_1 \leftrightarrow \mathbf{atp} \alpha \alpha' \in \Gamma_2) \rightarrow \{\Gamma_1 \vdash_n \langle \mathbf{atp} A B \rangle\} \rightarrow \{\Gamma_2 \vdash_n \langle \mathbf{atp} A B \rangle\}$.

H-Corollary 10 (D-Strengthening)

$$\{\mathbf{is_tp} \alpha :: \mathbf{atp} \alpha \alpha :: \Gamma \vdash_n \langle \mathbf{atp} A B \rangle\} \rightarrow \{\mathbf{atp} \alpha \alpha :: \Gamma \vdash_n \langle \mathbf{atp} A B \rangle\}.$$

H-Theorem 11 (Admissibility of Reflexivity for Types, G Version)

$$\mathbf{atpG} \Phi_{atp} \rightarrow \{\Phi_{atp} \vdash_n \langle \mathbf{is_tp} A \rangle\} \rightarrow \{\Phi_{atp} \vdash_n \langle \mathbf{atp} A A \rangle\}.$$

Proving reflexivity of equality for terms of the polymorphic lambda-calculus requires promotion of the corresponding lemma for types (Lemma 12), whose proof requires *c-str* and *c-wk*. These rules are expressed in Felty et al (2014, Section 2.3) using explicit strengthening functions (see Property 4). The particular function used here is $\text{rm}_{\text{is_tm } x; \text{aeq } x}^*$, and in Hybrid, we define it directly as a Coq recursive function.

$$\begin{aligned} \text{Fixpoint } \text{rm_aeq2atp} &: \text{atm } list \rightarrow \text{atm } list := \\ \text{rm_aeq2atp } (\text{is_tp } \alpha :: \text{atp } \alpha' \alpha'' :: \Phi_{\text{aeq}}) &= \\ & \quad (\text{is_tp } \alpha :: \text{atp } \alpha' \alpha'' :: \text{rm_aeq2atp } \Phi_{\text{aeq}}) \\ \text{rm_aeq2atp } (\text{is_tm } _ :: \text{aeq } _ _ :: \Phi_{\text{aeq}}) &= (\text{rm_aeq2atp } \Phi_{\text{aeq}}) \\ \text{rm_aeq2atp } _ &= \text{nil}. \end{aligned}$$

Note that this strengthening function transforms a context having schema aeqG (see Appendix A.2) to one having schema aeqG by removing an entire alternative and leaving the other intact. The third clause of this definition is required since this function can be called with any $(\text{atm } list)$, though this clause is never used since the function is only called on contexts having schema aeqG . This restriction is satisfied, for example, in the statement of the following lemma, which directly relates contexts satisfying these two schemas.

H-Lemma 12 $\text{aeqG } \Phi_{\text{aeq}} \rightarrow \text{atpG } (\text{rm_aeq2atp } \Phi_{\text{aeq}})$.

This result was left implicit in the informal proof. It is proved by a simple induction on the definition of aeqG . We then prove the required *c-str* and *c-wk* lemmas.

H-Lemma 13

1. $(\forall \alpha. \text{is_tp } \alpha \in \Gamma_1 \iff \text{is_tp } \alpha \in \Gamma_2) \rightarrow \{\Gamma_1 \vdash_n \langle \text{is_tp } A \rangle\} \rightarrow \{\Gamma_2 \vdash_n \langle \text{is_tp } A \rangle\}$.
2. $\text{aeqG } \Phi_{\text{aeq}} \rightarrow (\forall \alpha. \text{is_tp } \alpha \in \Phi_{\text{aeq}} \iff \text{is_tp } \alpha \in (\text{rm_aeq2atp } \Phi_{\text{aeq}}))$.
3. $\text{aeqG } \Phi_{\text{aeq}} \rightarrow (\forall \alpha, \alpha'. \text{atp } \alpha \alpha' \in \Phi_{\text{aeq}} \iff \text{atp } \alpha \alpha' \in (\text{rm_aeq2atp } \Phi_{\text{aeq}}))$.

Proof Note that (1) is similar to H-Lemma 9. (2) and (3) follow from a simple induction and the definition of rm_aeq2atp .

H-Corollary 14 (C-Strengthening/Weakening)

1. $\text{aeqG } \Phi_{\text{aeq}} \rightarrow \{\Phi_{\text{aeq}} \vdash_n \langle \text{is_tp } A \rangle\} \rightarrow \{(\text{rm_aeq2atp } \Phi_{\text{aeq}}) \vdash_n \langle \text{is_tp } A \rangle\}$
2. $\text{aeqG } \Phi_{\text{aeq}} \rightarrow \{(\text{rm_aeq2atp } \Phi_{\text{aeq}}) \vdash_n \langle \text{atp } A B \rangle\} \rightarrow \{\Phi_{\text{aeq}} \vdash_n \langle \text{atp } A B \rangle\}$

Proof (1) is a corollary of H-Lemma 13 (1) and (2), and (2) is a corollary of H-Lemma 9 and H-Lemma 13 (3).

Now, we can state and prove the required promotion lemma. We give the proof explicitly, though the formalized proof is fully automatic once the above lemmas are added to the hint database.

H-Lemma 15 (Promotion)

$\text{aeqG } \Phi_{\text{aeq}} \rightarrow \{\Phi_{\text{aeq}} \vdash_n \langle \text{is_tp } A \rangle\} \rightarrow \{\Phi_{\text{aeq}} \vdash_n \langle \text{atp } A A \rangle\}$.

Proof

$\text{atpG } (\text{rm_aeq2atp } \Phi_{\text{aeq}})$	by H-Lemma 12
$\{(\text{rm_aeq2atp } \Phi_{\text{aeq}}) \vdash_n \langle \text{is_tp } A \rangle\}$	by H-Corollary 14 (1)
$\{(\text{rm_aeq2atp } \Phi_{\text{aeq}}) \vdash_n \langle \text{atp } A A \rangle\}$	H-Theorem 11
$\{\Phi_{\text{aeq}} \vdash_n \langle \text{atp } A A \rangle\}$	by H-Corollary 14 (2)

Now, turning to the proof of admissibility of reflexivity for terms, in addition to promotion, we also need 2 membership lemmas and 2 *d-str* lemmas. We omit them here. The latter are needed in the ae_l and ae_{tl} cases, and are similar to H-Corollary 4 and H-Corollary 10.

H-Theorem 16 (Admissibility of Reflexivity for Terms, G Version)

$$\mathbf{aeqG} \Phi_{aeq} \rightarrow \{\Phi_{aeq} \vdash_n \langle \mathbf{is_tm} M \rangle\} \rightarrow \{\Phi_{aeq} \vdash_n \langle \mathbf{aeq} M M \rangle\}.$$

Proof As in the informal proof in the companion paper of Theorem 13, we show only the case for application of terms to types.

Case **ae.ta**: We know that M has the form $(\mathbf{tapp} M' A)$ and we must show:

$$\frac{\mathbf{h}_1 : IH \quad \mathbf{h}_2 : \mathbf{aeqG} \Phi_{aeq} \quad \mathbf{h}_3 : \{\Phi_{aeq} \vdash_n \langle \mathbf{is_tm} (\mathbf{tapp} M' A) \rangle\}}{\{\Phi_{aeq} \vdash_n \langle \mathbf{aeq} (\mathbf{tapp} M' A) (\mathbf{tapp} M' A) \rangle\}}$$

By repeated inversion of the SL rules on \mathbf{h}_3 , and repeated backward application of these rules to the conclusion, the above goal reduces to the following 2 subgoals:

$$\frac{IH \quad \mathbf{aeqG} \Phi_{aeq} \quad \{\Phi_{aeq} \vdash_{n-2} \langle \mathbf{is_tm} M' \rangle\} \quad \{\Phi_{aeq} \vdash_{n-2} \langle \mathbf{is_tp} A \rangle\}}{1. \{\Phi_{aeq} \vdash_{n-2} \langle \mathbf{aeq} M' M' \rangle\} \\ 2. \{\Phi_{aeq} \vdash_{n-2} \langle \mathbf{atp} A A \rangle\}}$$

The first is provable by *IH* and the second by promotion (H-Lemma 15).

6.2.2 R Version

The context relations $(\mathbf{atpR} \Phi_\alpha \Phi_{atp})$ and $(\mathbf{aeqR} \Phi_{\alpha x} \Phi_{aeq})$ are defined as Coq inductive predicates implementing **atpR** and **aeqR** from the **Definitions** section of Appendix A.2, similar to the inductive definition in Section 6.1.1 implementing **xaR** from Appendix A.1. Just as in the Hybrid definition of **xaR**, we remove the term and type well-formedness annotations from the second argument to the relations.

As before, *d-str* lemmas are not needed in the R version of admissibility of reflexivity, and membership lemmas are similar to H-Lemma 1. Admissibility for types is stated below with proof omitted; it is similar to those already shown.

H-Theorem 17 (Admissibility of Reflexivity for Types, R Version)

$$\mathbf{atpR} \Phi_\alpha \Phi_{atp} \rightarrow \{\Phi_\alpha \vdash_n \langle \mathbf{is_tp} A \rangle\} \rightarrow \{\Phi_{atp} \vdash_n \langle \mathbf{atp} A A \rangle\}.$$

In order to implement relational strengthening Lemma 15 (see Section 2.2) in Hybrid, we witness Φ_α and Φ_{atp} directly by applying strengthening functions to $\Phi_{\alpha x}$ and Φ_{aeq} , respectively. These functions are implemented similarly to **rm.aeq2atp** in the G version (Section 6.2.1). We only show the main clauses of the definitions.

$$\begin{aligned} \mathbf{rm_alphx2alph} (\mathbf{is_tp} \alpha :: \Phi_{\alpha x}) &= (\mathbf{is_tp} \alpha :: \mathbf{rm_alphx2alph} \Phi_{\alpha x}) \\ \mathbf{rm_alphx2alph} (\mathbf{is_tm} _ :: \Phi_{\alpha x}) &= (\mathbf{rm_alphx2alph} \Phi_{\alpha x}) \\ \mathbf{rm_aeq2atp} (\mathbf{atp} \alpha' \alpha'' :: \Phi_{aeq}) &= (\mathbf{atp} \alpha' \alpha'' :: \mathbf{rm_aeq2atp} \Phi_{aeq}) \\ \mathbf{rm_aeq2atp} (\mathbf{aeq} _ _ :: \Phi_{aeq}) &= (\mathbf{rm_aeq2atp} \Phi_{aeq}) \end{aligned}$$

The former transforms a context having schema **axG** to one having schema **aG**, and the latter transforms a context having schema **aeqG** to one having schema **atpG**. Using these functions, relational strengthening is stated as follows.

H-Lemma 18 (Relational Strengthening)

$$\mathbf{aeqR} \Phi_{\alpha x} \Phi_{aeq} \rightarrow \mathbf{atpR} (\mathbf{rm_alphx2alph} \Phi_{\alpha x}) (\mathbf{rm_aeq2atp} \Phi_{aeq}).$$

Note that this lemma is similar to H-Lemma 12 of the G version. We next state the required *c-str* and *c-wk* lemmas.

H-Corollary 19 (C-Strengthening/Weakening)

1. $\mathbf{aeqR} \Phi_{\alpha x} \Phi_{aeq} \rightarrow \{\Phi_{\alpha x} \vdash_n \langle \mathbf{is_tp} A \rangle\} \rightarrow \{(\mathbf{rm_alphx2alph} \Phi_{\alpha x}) \vdash_n \langle \mathbf{is_tp} A \rangle\}$.
2. $\mathbf{aeqR} \Phi_{\alpha x} \Phi_{aeq} \rightarrow \{(\mathbf{rm_aeq2atp} \Phi_{aeq}) \vdash_n \langle \mathbf{atp} A B \rangle\} \rightarrow \{\Phi_{aeq} \vdash_n \langle \mathbf{atp} A B \rangle\}$.

This result is stated as a corollary since it follows from the same kinds of lemmas (omitted here) as H-Corollary 14. The following promotion lemma follows from the above lemmas in a similar series of steps to the proof of Lemma 15.

H-Lemma 20 (Promotion)

$$\mathbf{aeqR} \Phi_{\alpha x} \Phi_{aeq} \rightarrow \{\Phi_{\alpha x} \vdash_n \langle \mathbf{is_tp} A \rangle\} \rightarrow \{\Phi_{aeq} \vdash_n \langle \mathbf{atp} A A \rangle\}.$$

Unlike H-Lemma 15, this proof cannot be fully automated in Coq using `eauto` because there are too many existential variables to fill in. In general, there are fewer when a single generalized context is used.

We now state the final result, admissibility of reflexivity for terms, whose proof is similar to the proof of H-Theorem 16, except there is no need for *d-str*.

H-Theorem 21 (Admissibility of Reflexivity for Terms, R Version)

$$\mathbf{aeqR} \Phi_{\alpha x} \Phi_{aeq} \rightarrow \{\Phi_{\alpha x} \vdash_n \langle \mathbf{is_tm} M \rangle\} \rightarrow \{\Phi_{aeq} \vdash_n \langle \mathbf{aeq} M M \rangle\}.$$

In this subsection, we have shown both the G and R versions in detail, to make explicit all the required lemmas (e.g., c- and d-strengthening and weakening, and promotion) and to illustrate that Hybrid is flexible enough to do different versions of this proof. In doing so, we follow the informal proofs very closely. If the goal were to just complete any proof and move on, we would have chosen the R version, since it is slightly simpler. In fact, an even simpler R version is done in Abella in Section 7.2, and that proof is also possible, and perhaps preferable in Hybrid.¹⁷

6.3 Non-linear Context Extensions

For the completeness of equality benchmark, the definitions of the Hybrid constants include `app` and `lam` and the definition of `atm` includes predicates `is_tm`, `aeq`, and `deq`. (See page 30 and Appendix A.1.) As mentioned in Section 5, the encoding of the inference rules for `deq` require some well-formedness annotations in order to prove adequacy. We briefly discuss the statements and proofs of internal adequacy at the end of this subsection after illustrating the R version of the completeness theorem. The G version appears in Appendix B.1. As usual, its proof is more complicated than the R version.

¹⁷ The version presented there (A-Theorem 8) illustrates the trade-off between choosing the smallest context relation possible for each lemma (as is done here), and choosing one that leads to a simpler but less modular proof development. There are also other proof developments described in Section 7 that diverge from the informal versions in the companion paper, which would also present no problems in Hybrid (e.g., A-Theorem 15).

6.3.1 R Version

The Hybrid definition of $(\text{adR } \Phi_a \Phi_{xd})$ implements adR in Appendix A.1. The first argument Φ_a does not contain the term well-formedness annotation, but Φ_{xd} does. We once again formalize relational strengthening (see H-Lemma 18) using a strengthening function rm_xd2x , whose main clause is:

$$\text{rm_xd2x } (\text{is_tm } x :: \text{deq } _ _ :: \Phi_{xd}) = (\text{is_tm } x :: \text{rm_xd2x } \Phi_{xd})$$

Unlike in Section 6.2 where all the strengthening functions removed an alternative from a context schema, this one relates two schemas each with just one alternative and modifies every block by removing one of the two atoms.

H-Lemma 22 (Relational Strengthening)

1. $\text{adR } \Phi_a \Phi_{xd} \rightarrow \text{xaR } (\text{rm_xd2x } \Phi_{xd}) \Phi_a$.
2. $\text{adR } \Phi_a \Phi_{xd} \rightarrow \text{aG } \Phi_a$.

In the above lemma, the first statement is the Hybrid version of Lemma 20 in Section 2.2, needed to prove promotion for reflexivity (Lemma 21). The promotion lemmas for symmetry and transitivity have much simpler statements and proofs, and were omitted from the informal proof. Here we make them explicit. Their proofs require the second statement above, but other than that are simple.

Promotion for reflexivity requires one *c-str* lemma, as follows.

H-Lemma 23 (C-Strengthening)

$$\text{adR } \Phi_a \Phi_{xd} \rightarrow \{\Phi_{xd} \vdash_n \langle \text{is_tm } A \rangle\} \rightarrow \{(\text{rm_xd2x } \Phi_{xd}) \vdash_n \langle \text{is_tm } A \rangle\}.$$

H-Lemma 24 (Promotion)

1. $\text{adR } \Phi_a \Phi_{xd} \rightarrow \{\Phi_{xd} \vdash_n \langle \text{is_tm } M \rangle\} \rightarrow \{\Phi_a \vdash_n \langle \text{aeq } M M \rangle\}$.
2. $\text{adR } \Phi_a \Phi_{xd} \rightarrow \{\Phi_a \vdash_n \langle \text{aeq } M N \rangle\} \rightarrow \{\Phi_a \vdash_n \langle \text{aeq } N M \rangle\}$.
3. $\text{adR } \Phi_a \Phi_{xd} \rightarrow \{\Phi_a \vdash_n \langle \text{aeq } M L \rangle\} \rightarrow \{\Phi_a \vdash_n \langle \text{aeq } L N \rangle\} \rightarrow \{\Phi_a \vdash_n \langle \text{aeq } M N \rangle\}$.

(In contrast, the G version of the above lemma in the appendix requires several more strengthening and weakening lemmas.)

H-Theorem 25 (Completeness, R Version)

$$\text{adR } \Phi_a \Phi_{xd} \rightarrow \{\Phi_{xd} \vdash_n \langle \text{deq } M N \rangle\} \rightarrow \{\Phi_a \vdash_n \langle \text{aeq } M M \rangle\}.$$

Proof The proof is by induction on n as usual. We discuss the same cases as in the informal proof of Theorem 22.

Case **de_r**: In the case when the last rule in the proof of $\{\Phi_{xd} \vdash_n \langle \text{deq } M N \rangle\}$ is reflexivity, we know that $M = N$ and $\{\Phi_{xd} \vdash_{n-1} \langle \text{is_tm } M \rangle\}$. From height weakening (part (a) of SL-Theorem 1), we also have $\{\Phi_{xd} \vdash_n \langle \text{is_tm } M \rangle\}$. We can then apply H-Lemma 28 (1) to obtain the desired result.

Case **de_t**: We know that there is an L such that $\{\Phi_{xd} \vdash_{n-1} \langle \text{deq } M L \rangle\}$ and $\{\Phi_{xd} \vdash_{n-1} \langle \text{deq } L N \rangle\}$. By the induction hypothesis, followed by height weakening, we can conclude $\{\Phi_a \vdash_n \langle \text{aeq } M L \rangle\}$ and $\{\Phi_a \vdash_n \langle \text{aeq } L N \rangle\}$. We can then apply H-Lemma 28 (3) to obtain the desired result.

Case **de_l**: This case is similar to other abstraction cases, such as the **tm_l** case of H-Theorem 2 except that height weakening is needed. We know that M and N have the form $(\text{lam } x. M'x)$ and $(\text{lam } x. N'x)$, respectively, and we must show:

$$\frac{\mathbf{h}_1 : IH \quad \mathbf{h}_2 : \mathbf{adR} \Phi_a \Phi_d \quad \mathbf{h}_3 : \{\Phi_d \vdash_n \langle \mathbf{deq} (\mathbf{lam} x. M'x) (\mathbf{lam} x. N'x) \rangle\}}{\{\Phi_a \vdash_n \langle \mathbf{aeq} (\mathbf{lam} x. M'x) (\mathbf{lam} x. N'x) \rangle\}}$$

By repeated inversion of the SL rules on \mathbf{h}_3 , and repeated backward application of these rules to the conclusion, the above goal reduces to the following subgoal:

$$\frac{IH \quad \mathbf{adR} \Phi_a \Phi_d \quad \mathbf{h}_4 : \mathbf{proper} x}{\mathbf{h}_5 : \{(\mathbf{is_tm} x :: \mathbf{deq} x x :: \Phi_{da}) \vdash_{n-4} \langle \mathbf{deq} (M'x) (N'x) \rangle\}} \\ \{(\mathbf{aeq} x x :: \Phi_{da}) \vdash_{n-3} \langle \mathbf{aeq} (M'x) (N'x) \rangle\}$$

The heights of \mathbf{h}_5 and the conclusion differ because the $\mathbf{de.l}$ rule has the additional well-formedness premise and the $\mathbf{ae.l}$ does not. We simply apply SL-Theorem 1 (a) to obtain $n - 3$ as the height for both and apply IH .

6.3.2 Internal Adequacy

Although internal adequacy results were not needed to complete the above proof, they are important, as already discussed. Here, we state the R version of adequacy for the \mathbf{aeq} judgment.

H-Theorem 26 (Internal Adequacy of Algorithmic Equality)

$$\mathbf{xaR} \Phi_x \Phi_a \rightarrow \{\Phi_a \vdash_n \langle \mathbf{aeq} M N \rangle\} \rightarrow \{\Phi_x \vdash_n \langle \mathbf{is_tm} M \rangle\} \wedge \{\Phi_x \vdash_n \langle \mathbf{is_tm} N \rangle\}.$$

Although such results often require their own versions of helper lemmas such as membership and $c\text{-str}$, their proofs present no further complications. G versions of adequacy can also be stated and proved, again presenting no further complications. Similar internal adequacy statements have been proved for all other benchmarks formalized in Hybrid.

6.4 Order

As mentioned in Section 5, exchange is covered by SL-Theorem 1 (b), which expresses weakening, contraction, and exchange. Theorem 23 in Section 2.2, however, cannot be formalized in Hybrid, not because of any problem with handling exchange, but because specialized substitution lemmas of this particular form cannot be proved in Hybrid. The proof shares some similarities with the proof of type preservation for parallel reduction, which also cannot be formalized in the current version of Hybrid. We illustrate the problem in the Appendix B.2.

6.5 Uniqueness

For the Hybrid proof of type uniqueness (Theorem 24 in Section 2.2), we adapt the one from Felty and Momigliano (2009). The definitions of the Hybrid constants used in this subsection include \mathbf{app} and \mathbf{lam} , where the latter now takes two arguments with the first representing the type of the bound variable. (See the **Syntax** section in Appendix A.3.) The definition of \mathbf{atm} includes predicates $\mathbf{is_tm}$ and \mathbf{oft} . The types are those of the simply-typed lambda-calculus, and thus there are no operators with binders. We take advantage of the underlying RL, and define types

inductively in Coq, and redefine the `cLAM` constant and `lam` constructor to take into account the type of the bound variable.

```

Inductive is_tp := i : is_tp | arr : is_tp → is_tp → is_tp.
Inductive con  := cAPP : con | cLAM : is_tp → con.
Definition lam := λM, A. APP (CON (cLAM A)) (LAM M).

```

Uniqueness of variables in a context does not hold if we consider the specification of the SL alone. In particular, recall the specification of the `sl_all` rule introduced on page 29. In this rule x represents an arbitrary term of type `expr`, and we cannot assume that it is a variable. For the special case of the type uniqueness benchmark, we showed in Felty and Momigliano (2009) that we can achieve uniqueness of context variables with only a surprisingly minimal amount of additional formal infrastructure. In particular, we used Hybrid’s `VAR` constructor to encode free variables, and added a definition (`nvC`) that provided the capability of creating a new variable that is *fresh* with respect to a context. In particular (`VAR v`) represents a variable, where v is a natural number, and (`nvC Φ`) returns the number that is one greater than the maximum of all the numbers representing free variables in Φ . Very little overhead is required, namely a small library of simple lemmas about `nvC` (see Felty and Momigliano, 2009). With this extra infrastructure, we cannot use the definition of the `xtG` schema in Appendix A.3 as directly as in the other benchmarks. In particular, the main clause of the inductive definition of `xtG` is:

$$\text{xtG } \Phi_t \rightarrow \text{xtG } ((\text{oft } (\text{VAR } (\text{nvC } \Phi_t)) A) :: \Phi_t)$$

Note that definition omits well-formedness annotations from the context as in several other benchmarks. Then, we need 3 membership lemmas. The first two rule out non-variables from the context, and the third expresses the required uniqueness constraints.

H-Lemma 27 (Context Membership)

1. $\text{xtG } \Phi_t \rightarrow (\text{oft } (\text{lam } B M) A) \notin \Phi_t$
2. $\text{xtG } \Phi_t \rightarrow (\text{oft } (\text{app } M N) A) \notin \Phi_t$
3. $\text{xtG } \Phi_t \rightarrow (\text{oft } M A \in \Phi_t) \rightarrow (\text{oft } M B \in \Phi_t) \rightarrow \exists i. (M = (\text{VAR } i) \wedge A = B)$.

H-Theorem 28 (Type Uniqueness)

$$\text{xtG } \Phi_t \rightarrow \{\Phi_t \vdash_n \langle \text{oft } M A \rangle\} \rightarrow \{\Phi_t \vdash_m \langle \text{oft } M B \rangle\} \rightarrow A = B.$$

Proof By complete induction on n as usual. Inversion on $\{\Phi_t \vdash_n \langle \text{oft } M A \rangle\}$ breaks the proof into cases as usual. In the variable case, we know $(\text{oft } M A) \in \Phi_t$, and by Lemma 31 (3) that M has the form $(\text{VAR } v)$. We can then apply inversion to the second sequent $\{\Phi_t \vdash_m \langle \text{oft } (\text{VAR } v) B \rangle\}$. Since there are no `prog` clauses for the variable case, by inversion we also conclude $(\text{oft } (\text{VAR } v) B) \in \Phi_t$, and thus by Lemma 31 (3), that $A = B$.

Case `of_a`: After several inversions on the first sequent, we get:

$$\begin{array}{l}
\text{h}_1 : IH \quad \text{h}_2 : \text{xtG } \Phi_t \quad \text{h}_3 : \{\Phi_x \vdash_{n-2} \langle \text{oft } M_1 (\text{arr } A' A) \rangle\} \\
\text{h}_4 : \{\Phi_x \vdash_{n-2} \langle \text{oft } M_2 A' \rangle\} \quad \text{h}_5 : \{\Phi_x \vdash_m \langle \text{oft } (\text{app } M_1 M_2) B \rangle\} \\
\hline
A = B
\end{array}$$

At this point, inversion on the h_5 results in two subcases. One is similar to the application cases of other proofs and can be completed by the induction hypothesis. The other introduces the new hypothesis $(\text{oft } (\text{app } M_1 M_2) B) \in \Phi_t$, which we rule out by Lemma 31 (2). The `of_l` case makes a similar use of Lemma 31 (1).

6.6 Substitution

The substitution lemma used in the type preservation for parallel reduction benchmark (Theorem 25 in Section 2.2) can be proven directly in Hybrid where parametric substitution is β -reduction and hypothetical substitution corresponds to an application of SL-Theorem 1 (c). Since we do not need uniqueness of variables, we redefine (the main clause of the definition of) context xtG and prove a simple but necessary lemma:

$$\text{xtG } \Phi_t \rightarrow \text{xtG } ((\text{oft } x A) :: \Phi_t)$$

H-Lemma 29 $\text{xtG } \Phi_t \rightarrow \{\Phi_t \vdash \langle \text{oft } M A \rangle\} \rightarrow \text{proper } M$.

H-Lemma 30 (Substitution)

$$\text{xtG } \Phi_t \rightarrow (\forall x. \text{proper } x \rightarrow \{(\text{oft } x A :: \Phi_t) \vdash_n \langle \text{oft } (M x) B \rangle\}) \rightarrow \{\Phi_t \vdash_m \langle \text{oft } N A \rangle\} \rightarrow \{\Phi_t \vdash_{n+m} \langle \text{oft } (M N) B \rangle\}.$$

The proof follows closely the informal proof of Lemma 25 in the companion paper.

As mentioned earlier, type preservation for parallel reduction as we have defined it here cannot be proved in the current version of Hybrid. This is an example where uniqueness of variables in the context is required, but the use of nvC as in the previous example is not enough. We refer the reader to Appendix B.2 for the details. Fixing this problem is the subject of current work in Hybrid. One approach is to use a form of “weak” HOAS that replaces the `s_all` rule (see page 29) with one that explicitly adds variables (using the `VAR` constructor) to the context. A better approach, which would not require sacrificing the benefits of full HOAS, is to implement the ∇ -quantifier in Hybrid.

7 Abella

Abella’s RL (\mathcal{G}) is an intuitionistic first-order sequent calculus endowed with a fixed-point approach to (co)induction and the nabla quantifier ∇ to support reasoning over λ -terms syntax. We refer to Gacek et al (2012, 2011) for a detailed account of Abella and its meta-theory and stick to the bare minimum.

Differently from Hybrid, the Abella system implements the two-level approach in a way that limits syntactic overhead. The system accepts `prog` clauses in the form of (executable) λ -Prolog files (Miller and Nadathur, 2012), analogously to how Beluga interacts with LF signatures. For example, the concrete syntax for algorithmic equality is:

$$\begin{aligned} \text{aeq } (\text{app } M1 M2) (\text{app } N1 N2) &: - \text{aeq } M1 N1, \text{aeq } M2 N2. \\ \text{aeq } (\text{lam } M) (\text{lam } N) &: - \text{pi } x \backslash \text{aeq } x x \Rightarrow \text{aeq } (M x) (N x). \end{aligned}$$

Abella hides details such as the height of derivations by displaying `seq` predicates with syntactic sugar and using an annotation-based treatment of induction. Abella also embeds meta-theoretic properties of the specification logic into *tactics* named `monotone`, `cut`, `inst`, which corresponds respectively to weakening (including exchange and contraction), cut and specialization in SL-Theorem 1. Generic judgments are handled by the nabla quantifier, i.e. the SL rule for universal quantification is:

$$\nabla x. \{\Gamma \vdash G x\} \rightarrow \{\Gamma \vdash \forall x. G x\}$$

We recall that nabla ensures freshness and correct scoping both at the left and right of the sequent, while recovering the usual instantiation properties of universal quantification at the SL level ensured by the *specialization* rule.

As we have mentioned before, Abella 2.0 offers a n-ary SL, which generalized the one described in Section 5 by moving from a single left rule (bc) to three focused left rules for universal, implication and conjunction as encoded by the judgment $\Sigma; \Gamma, [F] \rightarrow_C A$. In other words, we are switching from the proof-theory underlying Prolog (Horn clauses), to that of λ Prolog, more precisely to its first-order hereditary Harrop formula fragment. However, for the set of benchmarks that we consider in this paper, the n-ary SL collapses to the previous one; we only note the different shape of the axiom rule (match) and the one replacing backchaining (focus), as they have a different look w.r.t. proof script's variable cases and the related member lemmas:

$$\frac{}{\Sigma; \Gamma, [A] \rightarrow_C A} \text{ match} \qquad \frac{F \in \Gamma \cup \mathcal{C} \quad \Sigma; \Gamma, [F] \rightarrow_C A}{\Sigma; \Gamma \rightarrow_C A} \text{ focus}$$

7.1 Basic Linear Context Extensions

7.1.1 R Version

We encode our first context relation ($\mathbf{xaR} \Phi_x \Phi_{xa}$) with an Abella inductive definition (keywords **Define** and $:=$) for the binary predicate \mathbf{xaR} over the built-in sort of (atomic) assumptions **olist**:

```
Define xaR : olist → olist → prop by
  xaR nil nil;
  nabla x, xaR (is_tm x :: Ts) (aeq x x :: As) := xaR Ts As.
```

Recall that the encoding of \mathbf{aeq} and hence of Φ_{xa} does not contain the atom $\mathbf{is_tm} \ x$, but the latter is needed to mimic induction over individuals, namely for the proof of A-Theorem 2 below.

We start our development with a prototypical member lemma, namely the formalization of Lemma 6, where \mathbf{member} is Abella's built-in predicate for " \in ".

A-Lemma 1 (Context Membership)

```
member_is_tm: forall Ts As F,
  xaR Ts As → member F Ts → exists M, F = is_tm M ∧ member (aeq M M) As.
```

Proof By induction on the second premise.

The proof of such lemmas is completely routine; unfortunately, the user has to state and reprove this kind of results every time, as no tactical language is available in Abella per se. Note also the difference with the statement of the same result in Hybrid (H-Lemma 1): while in Hybrid's atomic contexts the predicate is immediately visible, in Abella 2.0 it needs to be singled out for focusing by the focus rule.

We are now ready for our first proof, which we will report in Abella's original notation (inspired by Coq). From the next proof on, we will somewhat streamline it. We will also omit outermost universal quantifiers, unless there is some interesting quantifier alternation.

A-Theorem 2 (Admissibility of Reflexivity, R Version)

$$\text{xaR Ts As} \rightarrow \{\text{Ts} \vdash \text{is_tm M}\} \rightarrow \{\text{As} \vdash \text{aeq M M}\}.$$

Proof By induction on the derivation of the second premise. The induction hypothesis, here $\{\text{Ts} \vdash \text{is_tm M}\}^*$, is annotated with a star that is used to track the size of inductive arguments – we will see it in action in the `tm_a` case.

After fixing variables and assuming the antecedents, a call to the `case` tactic, which implements inversion, on $\{\text{Ts} \vdash \text{is_tm M}\}$ yields three cases:

Case “variable”:

Variables: As, Ts, M

IH : forall As Ts, xaR Ts As \rightarrow $\{\text{Ts} \vdash \text{is_tm M}\}^* \rightarrow \{\text{As} \vdash \text{aeq M M}\}$

H1 : xaR Ts As

H2 : $\{\text{Ts}, [\text{F}] \vdash \text{is_tm M}\}^*$

H3 : member F Ts

 $\{\text{As} \vdash \text{aeq M M}\}$

Here we appeal to A-Lemma 1 (concrete syntax apply `member_is_tm` to H1 H3.):

Variables: As, Ts, M, M'

IH : forall As Ts T, xaR Ts As \rightarrow $\{\text{Ts} \vdash \text{is_tm T}\}^* \rightarrow \{\text{As} \vdash \text{aeq M T}\}$

H1 : xaR Ts As

H2 : $\{\text{Ts}, [\text{is_tm M}'] \vdash \text{is_tm M}\}^*$

H3 : member (is_tm M') Ts

H4 : member (aeq M' M') As

 $\{\text{As} \vdash \text{aeq M M}\}$

A further inversion on H2 unifies M with M', in particular in assumption H4. Then we use `search` tactic, which coincides with Prolog search, to finish the proof. This implicitly uses first the `focus` and then the `match` rule in the SL.

Case tm.a:

Variables: As, Ts, M, N

IH : forall As Ts M, xaR Ts As \rightarrow $\{\text{Ts} \vdash \text{is_tm M}\}^* \rightarrow \{\text{As} \vdash \text{aeq M M}\}$

H1 : xaR Ts As

H3 : $\{\text{Ts} \vdash \text{is_tm N}\}^*$

H4 : $\{\text{Ts} \vdash \text{is_tm M}\}^*$

 $\{\text{As} \vdash \text{aeq (app M N) (app M N)}\}$

Although Abella's proofs by induction are carried out over the height of the derivation, the system does not show the numbers, as Hybrid does, rather it employs a form of abstract interpretation, whereby the annotation (*) indicates that the height has indeed decreased and that the IH can be safely applied. Let us do that to H1 H3 (apply IH to H1 H3) and to H1 H4 and conclude the case by `search`, which in this case will entail selecting for focus (SL rule `prog`) clause `ae_a`.

Case tm.l:

Variables: As, Ts, T, M

IH : forall As Ts T, xaR Ts As \rightarrow $\{\text{Ts} \vdash \text{is_tm T}\}^* \rightarrow \{\text{As} \vdash \text{aeq M T}\}$

H1 : xaR Ts As

H3 : $\{\text{Ts}, \text{is_tm } x \vdash \text{is_tm (M } x)\}^*$

 $\{\text{As} \vdash \text{aeq (lam M) (lam M)}\}$

First, note that `x` is a fresh name, globally bound in the proof, generated by interpreting the object-level universal quantifier `pi` by ∇ . Now, we would like to apply

the inductive hypothesis as before. This is however allowed only if the contexts in H1 and H3 match, i.e., if $\text{xaR } (\text{is_tm } x :: \text{Ts}) (\text{aeq } x x :: \text{As})$ holds, see the appeal to rule crel_{xa} in the proof of Theorem 7. We can direct Abella to acknowledge this by the tactic `assertor` by using the underscore notation `apply IH to _ H3`, by which the system “guesses” the right context extension. Then we conclude by search.

What about symmetry (the same holds for transitivity, which we omit here)? If we are serious about the fine-grained view, we should state and prove it in the smallest possible world, namely the context predicate aG , which contains only assumptions of the form $\text{aeq } x x$. In any case, we first need another “member” lemma, stating an inversion property of the assumptions. This was implicitly enforced by rule aeq_v .

A-Lemma 3 (Context Inversion)

$\text{member_aeq_inv: aG As} \rightarrow \text{member F As} \rightarrow \text{exists M N, F = aeq M N} \wedge \text{M} = \text{N}$.

A-Theorem 4 (Admissibility of Symmetry)

$\text{aG As} \rightarrow \{\text{As} \vdash \text{aeq M N}\} \rightarrow \{\text{As} \vdash \text{aeq N M}\}$.

Proof By now standard induction on the second derivation, using A-Lemma 3 in the variable case.

If we want to “package up” the fact that algorithmic equality is an equivalence relation, we need to lift the above result(s) to the world where reflexivity is stated, namely xaR Ts As . In this case this is easy, since we can prove by a trivial induction (see also H-Lemma 26 (3)):

A-Lemma 5 $\text{aG_xaR_proj: xaR Ts As} \rightarrow \text{aG As}$

This lemma just makes explicit the fact that if two contexts (instances) are related, then they satisfy their respective schema. This is internalized in Beluga’s type theory, while it is a proof obligation in the much weaker-typed two level approach.

Thanks to this lemma we can lift the relevant property to the larger world.

A-Corollary 6 (Promotion of Symmetry)

$\text{xaR Ts As} \rightarrow \{\text{As} \vdash \text{aeq M N}\} \rightarrow \{\text{As} \vdash \text{aeq N M}\}$.

7.1.2 G Version

There seems to be no easy way to do a G-style proof in Abella 2.0, as opposed to previous versions, where one could exploit the simpler notion of contexts w.r.t. the second order SL.

Recall that the informal proof of reflexivity in Theorem 8 required a *d-strengthening* lemma, beyond *d-weakening*, stating that we can safely drop any `is_tm` assumption from a derivation of `aeq`, since the former judgment is irrelevant to the latter. See for example, in the Hybrid development, the proof of d-strengthening, H-Corollary 4, which follows from H-Lemma 3. Note that those results are stated for *any* (atomic) Γ , without any associated schema. However, once contexts can contain non-atomic assumptions, this result becomes obviously false, as this counterexample shows. If $c : \text{tm}$ we can prove:

```

nabla x, forall L K,
  L = (is_tm x  $\Rightarrow$  aeq x c) :: is_tm x :: aeq x x :: nil  $\rightarrow$ 
  K = (is_tm x  $\Rightarrow$  aeq x c) :: aeq x x :: nil  $\rightarrow$ 
    {L  $\vdash$  aeq x c}  $\wedge$  ({K  $\vdash$  aeq x c}  $\rightarrow$  false).

```

On the other hand, in Abella 2.0 we do have to state theorems over open terms under a context schema or relation, so that the shape of the context can be delineated in the member lemma. Hence, if we introduce the generalized context

```

Define xaG : olist  $\rightarrow$  prop by
  nabla x, xaG (is_tm x :: aeq x x :: Ga) := xaG Ga.

```

and state strengthening under this condition

```

A-Attempt 7 forall Ga M N, nabla x, xaG Ga  $\rightarrow$ 
  {Ga, aeq x x, is_tm x  $\vdash$  aeq (M x) (N x)}  $\rightarrow$ 
  {Ga, aeq x x  $\vdash$  aeq (M x) (N x)}.

```

the statement will not be provable, as the conclusion does not preserve the context schema. In particular, in the binding case, the context in the schema will not match the one passed to the SL judgment.

We can fix this particular benchmark by *changing* the specification of algorithmic equality so that it matches the generalized context, i.e.:

```

aeq (lam M) (lam N): - pi x \ is_tm x  $\Rightarrow$  aeq x x  $\Rightarrow$  aeq (M x) (N x).

```

However, this approach does not reasonably seem to scale elsewhere; consider the completeness benchmark, where we would be compelled to add spurious assumptions just to match the context schema. While this may not be the final word on G-style proofs in Abella 2.0, we see no point in investigating it further here, given also the clear superiority of R-proofs, even in the previous version of the system.

7.2 Linear Contexts Extensions with Alternative Declaration (R Version)

Here we depart from the methodological position we took in the companion paper, where we stated theorems in the smallest context predicate (relation) possible; we instead use here what could be termed the *least upper bound* of the relevant schemas, here `atpG` and `aeqG`, in this case the latter. This makes the development much smoother, since it does not require promotion lemmas (Lemma 16), for which, as we have mentioned, Abella provides no support. Note that the projection trick that we used before (A-Lemma 5) here does not apply. However, the “fine-grained” approach has been pursued in the Hybrid section and could be duplicated here, albeit more painfully, since strengthening functions would have to simulated relationally. Anyhow, one can argue that eventually algorithmic type equality will have to be packaged up with term equality and we may as well formulate all statements in the smallest *common* context to begin with.

Having gone this way, the development is completely uneventful. Once we state the expected context relation:

```

Define aeqR : olist  $\rightarrow$  olist  $\rightarrow$  prop by
  nabla x, aeqR (is_tm x :: Ts) (aeq x x :: As) := aeqR Ts As;
  nabla a, aeqR (is_tp a :: Ts) (atp a a :: As) := aeqR Ts As.

```

it is just a matter of proving the two standard member lemmas and moving directly to our version, which subsumes Theorem 14, Lemmas 15 and 16, and finally Theorem 17:

A-Theorem 8 (Admissibility of Reflexivity for Types and Terms)

1. $\text{aeqR } Ts \text{ As} \rightarrow \{Ts \vdash \text{is_tp } A\} \rightarrow \{As \vdash \text{atp } A \ A\}$.
2. $\text{aeqR } Ts \text{ As} \rightarrow \{Ts \vdash \text{is_tm } M\} \rightarrow \{As \vdash \text{aeq } M \ M\}$.

Proof

1. By induction on the second derivation.
2. By induction on the second derivation, using point (1) in the `tapp` case.

7.3 Non-Linear Context Extensions (R Version)

The encoding of the completeness proof in Abella turns out to be slightly more challenging. As we have seen in Subsection 4.2 of the companion paper, the encoding of declarative equality “suffers” from the need to reflect at the judgmental level the sorting statement $M : \text{tm}$. Recall in fact that in the reflexivity case of the mathematical proof of Theorem 22 from the assumption $\Phi_{xd} \vdash \text{deq } M \ M$ we were able to infer $\Phi_{xd} \vdash \text{is_tm } M$ by construction. In some cases, for example w.r.t. algorithmic equality, this invariant can be inductively established without further ado – note that this result is not needed in any of the following developments.

A-Theorem 9 (Internal Adequacy of Algorithmic Equality)

$$\text{xar } Ts \text{ As} \rightarrow \{As \vdash \text{aeq } M1 \ M2\} \rightarrow \{Ts \vdash \text{is_tm } M1\} \wedge \{Ts \vdash \text{is_tm } M2\}.$$

Proof Usual induction on $\{As \vdash \text{aeq } M1 \ M2\}$

An analogous result holds for declarative equality only due to the fact that we have strategically added `is_tm` annotations, as discussed in Section 5, page 28.

In these proofs two relations are at work: $(\text{dar } \Phi_{xa} \ \Phi_{xd})$ and $(\text{xar } \Phi_x \ \Phi_{xa})$, the former in the main statement, the latter in the relational strengthening Lemma 20. It significantly simplifies the implementation if we join them together in a *ternary* relation, which works as a “bridge” between the two—see the Hybrid section for a strictly fine-grained version more closely connected to the analysis in Section 3.2.2 of the companion paper:

Define `xadR` : `olist` \rightarrow `olist` \rightarrow `olist` \rightarrow **prop** **by**
`nabla` `x`, `xadR` (`is_tm` `x` :: `Ts`) (`aeq` `x` `x` :: `As`) (`deq` `x` `x` :: `is_tm` `x` :: `Ds`)
`:=` `xadR` `Ts` `As` `Ds`.

Let us now consider the statement of our theorem $\text{xadR } Ts \text{ As } Ds \rightarrow \{Ds \vdash \text{deq } M1 \ M2\} \rightarrow \{As \vdash \text{aeq } M1 \ M2\}$. First, we have to lift the equivalence properties of `aeq` to the current world `xadR`, easily accomplished analogously to what we have done in A-Lemma 5:

A-Lemma 10 `xar_xadR_proj`: $\text{xadR } Ts \text{ As } Ds \rightarrow \text{xar } Ts \text{ As}$

A-Lemma 11 (Promotion of Transitivity and Symmetry)

(Tr) $\text{xadR } Is \text{ As } Ds \rightarrow \{As \vdash \text{aeq } M1 \ M2\} \rightarrow \{As \vdash \text{aeq } M2 \ M3\} \rightarrow \{As \vdash \text{aeq } M1 \ M3\}$.

(Sy) $\text{xadR Ts As Ds} \rightarrow \{\text{As} \vdash \text{aeq M N}\} \rightarrow \{\text{As} \vdash \text{aeq N M}\}.$

This is not quite enough: consider the case *de_r* of the completeness proof:

$$\frac{\text{xadR Ts As Ds} \quad \{\text{Ds} \vdash \text{is_tm M2}\}^*}{\{\text{As} \vdash \text{aeq M2 M2}\}}$$

Here, in order to apply our reflexivity A-Theorem 2, we need to *c-strengthen* the assumption $\{\text{Ds} \vdash \text{is_tm M2}\}$ to $\{\text{Ts} \vdash \text{is_tm M2}\}$. First we state a by now familiar member lemma:

A-Lemma 12 $\text{member_deq: xadR Is As Ds} \rightarrow \text{member F Ds} \rightarrow$
 $(\text{exists M, (F = is_tm M} \vee \text{F = deq M M}) \wedge$
 $\text{member (aeq M M) As} \wedge \text{member (is_tm M) Is}).$

Although the statement is somewhat more involved, the proof is still elementary.

The proof of *c-strengthening* is actually not difficult thanks to the way Abella implements SL-weakening:

A-Lemma 13 $\text{is_tm_cstr: xadR Ts As Ds} \rightarrow \{\text{Ds} \vdash \text{is_tm M}\} \rightarrow \{\text{Ts} \vdash \text{is_tm M}\}.$

Proof By induction on the second derivation. The interesting case is the binder:

$$\frac{\dots \text{H3 : } \{\text{Ds, is_tm x} \vdash \text{is_tm (M x)}\}^*}{\{\text{Is} \vdash \text{is_tm (lam M)}\}}$$

Here we need to weaken H3 so as to apply the IH. The key insight is *not* to do it manually with `assert {Ds, is_tm x, deq x x} ⊢ is_tm (M x)`, because this would loose the `*` annotation and the IH would not be applicable anymore. Rather we directly use SL weakening with the tactic `monotone H3 with (deq x x :: is_tm x :: Ds)` yielding

$$\frac{\dots \text{H4 : } \{\text{Ds, is_tm x, deq x x} \vdash \text{is_tm (M x)}\}^*}{\{\text{Is} \vdash \text{is_tm (lam M)}\}}$$

Now the IH is applicable and the rest of the proof is routine.

A-Corollary 14 (Promotion of Reflexivity)

$$\text{xadR Ts As Ds} \rightarrow \{\text{Ds} \vdash \text{is_tm M}\} \rightarrow \{\text{As} \vdash \text{aeq M M}\}.$$

Proof By A-Theorem 2 and A-Lemma 13.

A-Theorem 15 (Completeness)

$$\text{xadR Ts As Ds} \rightarrow \{\text{Ds} \vdash \text{deq M1 M2}\} \rightarrow \{\text{As} \vdash \text{aeq M1 M2}\}.$$

Proof By induction on the second premise. We show a couple of cases:

Case de_r: By A-Corollary 14.

Case de_t:

$$\frac{\text{xadR Ts As Ds} \quad \{\text{Ds} \vdash \text{deq M1 M'}\}^* \quad \{\text{Ds} \vdash \text{deq M' M2}\}^*}{\{\text{As} \vdash \text{aeq M1 M2}\}}$$

Here we simply apply the IH twice and then appeal to our promoted version of transitivity, A-Lemma 11. Symmetry is analogous, while the lambda case is uneventful.

7.4 Order (R Version)

This case study brings in some more sophisticated reasoning about variable identities; it is therefore convenient here and in the sequel to introduce the following definition, which recognizes when a term is in fact a (unique) name:

Define `name` : `tm` \rightarrow `prop` by
`nabla x, name x.`

Doing so this late in the game shows that the examples so far did not use in any essential way the “uniqueness” assumption in context schemas. But this is going to change from now on. Hence we restate A-Lemma 3 with the additional conclusion that `N` and hence `M` are indeed names:

A-Lemma 16 `member_aeq_inv_name`: `aG As` \rightarrow `member F As` \rightarrow
`exists M N, F = aeq M N \wedge M = N \wedge name N.`

The next statement uses `nabla` to bind the spot where substitution will occur.

A-Theorem 17 (Pairwise Substitution)

forall `As M1 M2 N1 N2, nabla x, aG As` \rightarrow
`{As, aeq x x \vdash aeq (M1 x) (M2 x)}` \rightarrow `{As \vdash aeq N1 N2}` \rightarrow
`{As \vdash aeq (M1 N1) (M2 N2)}`.

Proof By induction on the second derivation. Remarkably, the only interesting case is the variable one:

```
{As  $\vdash$  aeq N1 N2}
H2: member (F x) (aeq x x :: As)
{As, aeq x x, [F x]  $\vdash$  aeq (M1 x) (M2 x)}*
-----
{As  $\vdash$  aeq (M1 N1) (M2 N2)}
```

Case analysis on `H2` yields two sub-cases: `aeq (M1 x) (M2 x) = (aeq x x)`, hence `M1, M2` are the identity functions and the goal reduces to `{As \vdash aeq N1 N2}`. The second sub-case is when `member (F x) As`. By A-Lemma 16 we know `M1 = M2` and `(M2 n)` is a name. This entails two additional possibilities. First, it could be a different name `y`, leading to the goal `{As y \vdash aeq y y}`, which follows immediately since we have `member (aeq y y) (As y)`. Alternatively, as before, `M1` could be the identity function and again the goal is `{As \vdash aeq N1 N2}`, which we have.

The binder case follows straightforwardly by induction, since the SL treats object level contexts modulo SL-weakening, that is as *sets*.

Were we, for whatever reason (see Momigliano, 2012, for a particular perspicuous instance), to treat contexts explicitly at the meta-level, that is, had `aeq` type `olist` \rightarrow `tm` \rightarrow `tm` \rightarrow `prop`, we would have to formalize exchange as an inductive relation, in a “structural” enough way to work well with Abella’s reasoning style; this is not completely trivial, as exhibited in the cited Momigliano (2012).

7.5 Uniqueness (R Version)

This benchmark is explained in full detail in Gacek et al (2012), so we keep it short. The schema definition

Define $\text{xtG} : \text{olist} \rightarrow \text{prop}$ by
 $\text{nabla } x, \text{xtG} (\text{oft } x \ A :: \text{Os}) := \text{xtG } \text{Os}.$

says that $\text{xtG } \text{Os}$ holds if and only if Os is a list of atoms $\text{oft } x \ A$ in which x is a nominal constant that does not appear elsewhere in the list; thus its typing must be unique, condition enforced by means of the correct alternation between nabla and the outermost universal quantification. Compare it to a nabla -less version:¹⁸

Define $\text{xtG}' : \text{olist} \rightarrow \text{prop}$ by
 $\text{xtG}' (\text{oft } X \ A :: \text{Os}) :=$
 $(\text{forall } M \ N, X = \text{app } M \ N \rightarrow \text{false}) \wedge$
 $(\text{forall } A \ M, X = \text{abs } A \ M \rightarrow \text{false}) \wedge$
 $(\text{forall } A', \text{member } (\text{oft } X \ A') \ \text{Os} \rightarrow \text{false}) \wedge \text{xtG}' \ \text{Os}.$

Here we must explicitly restrict structure and occurrences of X , as we have no way to internalize the fact that this is indeed a name.

Two member lemmas are required, one to rule out impossible cases when inverting on the typing relation:

$\text{member_of_name}: \text{xtG } \text{Os} \rightarrow \text{member } E \ \text{Os} \rightarrow \text{exists } M \ A, E = \text{oft } M \ A \wedge \text{name } M$

the other to prove the desired property for the variable case:

$\text{xtG_uq}: \text{xtG } \text{Os} \rightarrow \text{member } (\text{oft } M \ A) \ \text{Os} \rightarrow \text{member } (\text{oft } M \ A) \ \text{Os} \rightarrow A = B.$

A-Theorem 18 (Type Uniqueness)

$\text{xtG } \text{Os} \rightarrow \{\text{Os} \vdash \text{oft } M \ A1\} \rightarrow \{\text{Os} \vdash \text{oft } M \ A2\} \rightarrow A1 = A2.$

Proof By induction on $\{\text{Os} \vdash \text{oft } M \ A1\}$ and inversion on $\{\text{Os} \vdash \text{oft } M \ A2\}$, using xtG_uq in the variable case and member_of_name in the case analysis on $\{\text{Os} \vdash \text{oft } M \ A2\}$.

7.6 Substitution (R Version)

The context relation is unsurprising:

Define $\text{xrtR} : \text{olist} \rightarrow \text{olist} \rightarrow \text{prop}$ by
 $\text{nabla } x, \text{xrtR} (\text{pr } x \ x :: \text{Ps}) (\text{oft } x \ A :: \text{Os}) := \text{xrtR } \text{Ps } \text{Os}.$

We need two familiar member lemmas, the above member_of_name to clean up inversion plus a version of the inversion A-Lemma 3 for pr after which we can state the Abella version of Theorem 23:

A-Theorem 19 (Type Preservation for Parallel Reduction)

$\text{xrtR } \text{Ps } \text{Os} \rightarrow \{\text{Ps} \vdash \text{pr } M1 \ M2\} \rightarrow \{\text{Os} \vdash \text{oft } M1 \ A\} \rightarrow \{\text{Os} \vdash \text{oft } M2 \ A\}.$

Proof By induction on the derivation of $\{\text{Ps} \vdash \text{pr } M1 \ M2\}$ and inversion on $\{\text{Os} \vdash \text{oft } M1 \ A\}$. We show only two cases:

Case “variable”:

$$\frac{\text{xrtR } \text{Ps } \text{Os} \quad \{\text{Os} \vdash \text{oft } M1 \ A\} \quad \dots \quad \text{member } F \ \text{Ps}}{\text{Os} \vdash \text{oft } M2 \ A}$$

¹⁸ Recall that Hybrid uses the nvC function for new variables and H-Lemma 31 was required.

Here we use the inversion lemma to enforce that $M1 = M2$ – note that we do not need the fact that they are indeed names.

Case pr_b:

$$\frac{\{0s \vdash \text{oft} (\text{app} (\text{lam } A1) N1) T\} \quad \{Ps, \text{pr } n \ n \vdash \text{pr} (M1 \ n) (M2 \ n)\}^*}{\{Ps \vdash \text{pr } N1 \ N2\}^*}$$

$$\{0s \vdash \text{oft} (M2 \ N2) T\}$$

The non trivial inversion on the first premise yields the two assumptions H' : $\{0s \vdash \text{oft} (\text{lam } M1) (\text{arr } B \ A)\}$ and $\{0s \vdash \text{oft } N1 \ B\}$. By IH we have $\{0s \vdash \text{oft } N2 \ B\}$. We invert further on H' and get:

$$\dots H: \{Ps, \text{pr } n \ n \vdash \text{pr} (M1 \ n) (M2 \ n)\}^* \quad \{0s, \text{oft } n \ B \vdash \text{oft} (M1 \ n) A\}$$

$$\{0s \vdash \text{oft} (M2 \ N2) A\}$$

We apply the IH and have $\{0s, \text{oft } n \ B \vdash \text{oft} (M2 \ n) A\}$. The informal proof here applies the substitution lemma to get the desired conclusion. So we instantiate n in H with the appropriate term, namely $N2$ (tactic `inst`). Here we see that the nabla quantifier not only provides the correct scoping mechanism but also the required substitution behavior.

$$\{0s \vdash \text{oft } N2 \ B\} \quad \{0s, \text{oft } N2 \ B \vdash \text{oft} (M2 \ N2) A\}$$

$$\{0s \vdash \text{oft} (M2 \ N2) A\}$$

Now apply cut and it is all over.

8 Summary and Discussion

Systems supporting HOAS fall into two categories. On one side of the spectrum, we have systems built on type-theoretic foundation such as Twelf, an implementation of the logical framework LF, and the dependently-typed functional language Beluga. On the other side there are systems such as Abella and Hybrid that are based on a proof-theoretic foundation and follow the two-level approach, implementing a specification logic inside a higher-order logic or type theory.

8.1 Systems Based on Type Theory

Both Twelf and Beluga encode formal systems in the logical framework LF. In both systems d-weakening and d-exchange are built into the underlying type theory of LF, while d-strengthening is supported via subordination. Applying substitution lemmas (parametric and hypothetical) is simply β -reduction in these systems.

However, the two systems differ in their encoding of proofs. While Beluga provides a separate reasoning language where we are able to abstract over contexts of assumptions, in Twelf we remain within the logical framework LF and represent proofs as relations. This has several consequences: in Twelf, there is no simple way of inspecting a context and writing a generic variable case. All assumptions are implicit. Instead of a generic base case that applies to all variables in a given context, the proof for the base case is introduced whenever we encounter a variable

or assumption. As a consequence, we are assuming a given property we want to prove holds for variables, instead of proving the property for all variables. For example, in the uniqueness proof, we assume that every variable we introduce has a unique type instead of proving in general that whenever we introduce a variable to the context together with its type, it must be unique. World checking then ensures that for all variables an appropriate base case has also been introduced. Therefore, specifying worlds and checking that a given LF specification is consistent with it requires going outside of LF. In other words, while we write proofs as relations in LF, verifying that the relation constitutes a valid proof is accomplished by a variety of checkers (mode, coverage, termination, and world checkers) whose foundations lie outside of LF. Moreover, there is a conceptual gap between the on-paper proof, the relational representation of the proof in LF, and its interpretation as a total function, as established by the totality checker.

Proofs written as relations in Twelf live in an ambient context of assumptions and proving context membership lemmas such as Lemma 6 and Lemma 9 is unnecessary. The ambient context is the smallest general context containing all necessary hypothetical and parametric assumptions. World checking takes into account context subsumption and validates that we can refer to a relation defined in a “smaller” world (for example `ref1G`) while defining a relation in a “bigger” world (for example `ceqG`). Hence, promotion lemmas such as Lemma 12 and Lemma 18 are redundant. However, contexts are not first-class in Twelf; as such, we cannot abstractly reason about properties of contexts. Twelf also lacks inductive definitions, and for good reasons; thus it cannot express context relations directly. As a consequence, proofs via context relations would require a substantial overhead of encoding reifying assumptions explicitly (see for example Crary, 2009); it is unclear how to make the reasoning behind context subsumption (i.e., context strengthening combined with context weakening) explicit.

Beluga takes a different approach providing a separate language for implementing proofs. The proof language supports abstractions over contexts of assumptions and schemas classifying contexts. Moreover, we encapsulate a LF object together with its context to form a contextual object. Unlike Twelf’s global ambient context, every object in Beluga carries its own context in which it is meaningful. Beluga’s reasoning language also supports splitting on contextual objects and contexts and even defining relations about them.

This has three consequences: First, we are able to write a generic base case by pattern matching on elements in the contexts and abstracting over objects denoting variables; in particular, coverage checking (Dunfield and Pientka, 2009; Pientka et al, 2014) guarantees that given an object $[\gamma.M]$ of type $[\gamma.t\mathbf{m}]$, we know that either M is a variable from γ or an element constructed by the constants `lam` and `app`. Hence, proving context membership lemmas such as Lemma 6 and Lemma 9 explicitly is unnecessary when proving the G version of properties. Second, we inherit properties about abstract contexts such as uniqueness of declarations; we can also state properties about contexts using recursive types and move between contexts via substitutions; this gives the programmer considerable flexibility in how to implement proofs. Both the G version and the R version are possible, although the R version comes with a considerable overhead. Third, there is no conceptual gap between the on-paper proof and its implementation as a function;

the flow of information in the proof corresponds directly to the sequence of steps in the program and establishing correctness of the mechanization is more direct.

Type checking guarantees that we are manipulating well-formed contexts and well-formed derivations described using contextual objects and contexts. As in Twelf, type checking has declaration weakening built-in and supports declaration strengthening via subordination. Similarly, Beluga supports context subsumption, i.e., it uses subordination to justify safe context weakening and strengthening. Hence, promotion lemmas such as Lemma 12 and Lemma 18 are redundant. To reduce the trusted base, one may wish to prove these lemmas explicitly. (See our earlier discussion in Sec. 4.7.1.) Beluga’s language allows us to express context relations as indexed recursive datatypes (similar to inductive definitions) and hence the programmer can choose to make context strengthening and promotion explicit; this however significantly increases the proofs one needs to write and establish.

8.2 Systems Based on Proof Theory

Hybrid and Abella model hypothetical judgments of object logics using implication in the SL and parametric judgments via (generic) universal quantification. Hypothetical substitution is justified by appealing to the cut-admissibility lemma of the SL, i.e., via SL-Theorem 1 (c). Parametric substitution corresponds to the specialization rule of the corresponding meta-level universal quantifier, SL-Theorem 1 (d). Weakening is obtained directly from the specification logic, and so is exchange, via SL-Theorem 1 (b). Strengthening in these systems is proved on a case-by-case basis. There is so far no analogue to the subordination dependency analysis implemented in Twelf and Beluga, although it is conceivable to do so. Note that all weakening and strengthening lemmas are applied explicitly via either an SL theorem or a specialized strengthening theorem. In Hybrid, some of these lemma applications can be automated via hints to the prover.

In both Abella and Hybrid contexts are represented inductively as lists of assumptions. While the choice of using lists is somewhat arbitrary and other notions such as (multi)sets have been used (Hybrid) or are scheduled to be used (Abella), the focus on atoms is more deliberate as it is tied to how these systems establish that a context satisfies a given schema. Since these systems are untyped as far as contexts and schemas are concerned, the inductive specification of a context relation can be loosely seen as containing the specification of a schema. For an example, we refer to A-Lemma 5. The fact that a context *satisfies* a given schema then needs to be explicitly established as a theorem, easily proved via Prolog-like search in the meta-logic. However, this breaks down once we move beyond *atomic* assumptions, as we can in Abella 2.0, because the right introduction rule for inductive definitions does not work modulo “focusing” of the non-atomic assumption in the definition. This is more restrictive compared to Beluga, which naturally supports higher-order assumptions due to its general type-theoretic foundation.

With regard to the property of uniqueness of assumptions, Abella uses the ∇ -quantifier to abstract over variables, unlike the standard universal quantifier that abstracts over terms. In Hybrid there is an unresolved tension between the loose “forall” interpretation of terms in contexts and the use of the `nvC` function explained in Section 6. In the former case, schema satisfaction is not encoded adequately, since a Hybrid context (relation) definition will validate contexts with

non-variable terms or with terms not occurring uniquely. With the latter, contexts *are* adequately encoded and do occur uniquely, as seen in H-Lemma 31. However, we cannot always use this function, and even when we can, we lose the substitution property, see H-Attempt 20. In this regard, Abella’s support for the ∇ -quantifier is a real asset, as it provides both scoping and instantiation. Implementing the ∇ -quantifier in Hybrid (possibly following Gacek et al, 2011; Baelde, 2009) is a subject of current work, which we expect will solve this problem.

In both Hybrid and Abella, splitting and induction on inductive predicates is supported as part of the proof-theoretical foundation, but splitting and inducting directly on terms is not. As mentioned in Section 5, one of the main reasons for introducing in Hybrid and Abella the `is_tm` predicate is to provide induction at the level of syntax.

Historically Hybrid was the first “mainstream” implementation of the two-level architecture (Felty, 2002; Momigliano and Ambler, 2003), as compared to the tool used in McDowell and Miller (2002). Hybrid can be seen as an “opened-up” Abella, explicit, easy to experiment with, and to trust. The system permits very flexible proof development; both the generalized and the relational approach can be accommodated, although typically the relational approach will lead to more compact proofs, since fewer infrastructural lemmas need to be established.

Another difference between these two systems is the way that induction is used. In Hybrid proofs are by *complete* induction on the explicit height of derivations in the SL; this is quite time consuming for the user, who needs to provide the correct instantiation of heights of sub-derivations. At worst, the correct height can be a non trivial function of the given derivations. For example, in Section 2.2 of Felty and Pientka (2010), a `shape` judgment is defined that is weaker than `aeq`. In the electronic appendix to that paper, a context relation is defined and it is proved that if $\{\Phi_1 \vdash_n \langle \text{aeq } M \ N \rangle\}$ and Φ_1 is related to Φ_2 , then $\{\Phi_2 \vdash_{2n+2} \langle \text{shape } M \ N \rangle\}$. On the other hand, a proof by complete induction is just an application of Coq’s `lt_wf_ind` theorem and therefore easy to check and trust. In Abella the induction tactic is a very clever piece of code that hides all the boring details of height manipulation from the user, but at the same time it is somewhat opaque. This may become problematic when we have to manipulate an assumption before applying the (co)inductive hypothesis; for example, if we apply a user-proven lemma, say strengthening, even one that preserves the proof height, there is no way to convince the system that applying the hypothesis afterward does not violate the inductive restrictions. In those cases we have to change the specification so as to introduce another OL judgment to induct on, one which is not affected by said lemma.

From a theoretic point of view, Gacek sketches in his dissertation (?) how to convert proofs following the annotation-based schema for induction into ones using \mathcal{G} ’s original invariant-based induction rule. The consistency of the logic is shown by cut-elimination extending to \mathcal{G} the cut-elimination proof eventually published in ?.

Abella’s RL is quite weak, in the De Bruijn sense (de Bruijn, 1991), while Coq is among the strongest logical frameworks in existence. Beyond the obvious difference between simple and dependent types, which Hybrid does not really use that much, Hybrid inherits Coq’s full recursive function space, and that is a mixed blessing. It allows us to define functions (as opposed to the relational-only approach of Abella), which is quite handy, as exemplified with respect to promotion lemmas; at the same time, it brings back all the (external) adequacy issues that have been

studied in the last twenty years; hence the `abstr` annotations as explained in Felty and Momigliano (2012). A related issue is the different equality relations of the two systems. In Abella, it is simply $\alpha\beta\eta$ -conversion, while Hybrid again inherits Coq’s more complicated notion. This complication emerges, for example, in proofs by inversion, which in Hybrid sometimes require appeal to special equality and inversion lemmas.

Abella 2.0 is clearly more powerful than Hybrid as far as the SL is concerned because it implements a higher-order logic instead of a second-order one; nevertheless, this increased expressivity comes with some issues, to begin with the break down of schema satisfaction. More pragmatically Abella 2.0 is not conservative with respect to the previous “atoms-only” incarnation, in particular making the G-approach utterly problematic, which may not be a loss after all. It would be fairly easy to extend Hybrid’s SL to the n-ary case; in fact, this could be done *conservatively*, as there would be no problem in having both SLs coexist, for the user to choose according to the nature of the benchmark. However, we feel, and we realize this is a judgment call, that n-ary benchmarks do not occur so frequently in the “wild” to make this change a priority. Besides, most of the time those benchmarks can be easily brought back to the second-order case without too much sacrifice. There is also an additional Hybrid-specific issue: Hybrid’s abstraction mechanism `LAM` is intrinsically second-order, as so is the `abstr` annotation. While a theory of n-ary abstractions is feasible, it has not been developed yet. Therefore, even with a n-ary SL, there would be some benchmarks, such as any based on, say third-order constructs, e.g., `callcc`, that Hybrid would not be able to handle.

8.3 Further Discussion and Summary

Based on the proof developments and the discussion so far, we can see that both Beluga and Abella provide support beyond what we typically get with HOAS. By extending LF with first-class contexts and contextual types, Beluga provides intrinsic support for abstracting over variables and contexts, and relating them via first-class substitutions and inductive definitions. In Abella, the ∇ -quantifier abstracts over objects denoting variables, but not contexts and, unless they carry no information such as in Accattoli (2012), contexts still need to be handled explicitly. Although Hybrid’s lack of intrinsic support for variables increases the burden on the users compared to the other systems, it is the system with the richest type theory; we can readily take advantage of writing recursive functions in Coq instead of relations avoiding infrastructural lemmas about determinacy and totality of relations. However, we cannot yet write functions over Hybrid terms (`expr`). Hybrid can readily take advantage of Coq’s tactics, libraries, and decision procedures. Although the benchmarks we have considered in this paper do not showcase the need for libraries, they are of utmost importance in real-life specifications.

Both Hybrid and Beluga permit flexible proof development and both the generalized and the relational approach can be accommodated. In Hybrid proofs following the relational approach will be more compact, since fewer infrastructural lemmas need to be established and context strengthening can be encoded using recursive functions, thereby not appearing explicitly in the proof. In Beluga, proofs following the generalized approach will be more compact due to the support for weakening and context subsumption provided by the underlying type theory.

	Twelf	Beluga	Abella 2.0	Hybrid
OL ctx	implicit	implicit	implicit	implicit
RL ctx	implicit	explicit	list of formulas	list of atoms
schemas	block decl	Σ -type	inductive predicate	inductive type
schema sat	world checking	type checking $\Phi : S_x$	proof of $(xC \Phi)^\dagger$	proof of $(xC \Phi)$
d-wk	LF weakening	CMTT weakening	SL weakening	SL weakening
d-str	subordination \ddagger	subordination \ddagger	case by case \ddagger	case by case
c-wk/str	subordination \ddagger	subordination \ddagger	case by case \ddagger	case by case
d/c-exc	LF exc \ddagger	CMTT exc \ddagger	admissible in SL	admissible in SL
uniq	Assumed \ddagger	Parameter var.	nabla in the head	nvC function \ddagger
p subst	β -red	β -red	spec of ∇ in SL	spec of \forall in SL
h subst	β -red	β -red	cut in SL	atomic cut in SL \ddagger
G version	✓	✓	\ddagger	✓
R version	×	✓	✓	✓

\ddagger : may not always work, \ddagger : more aggressive than in Twelf

Table 1 At-a-Glance Features

With regard to trust, Hybrid may be thought of the most trustworthy system. Since Coq builds proofs in the form of CIC lambda-terms, a completed Hybrid proof can produce a Coq term that can be independently checked. In particular, all the splitting, inductive reasoning, including reasoning about proof heights, and all applications of context lemmas, including weakening, strengthening, membership, etc., will be present in the Coq term and amenable to Coq’s type checker.

Beluga’s proof is a program which can be checked independently by a small type checker which guarantees that contexts and contextual objects are well-formed; however, coverage is outside of the type checker and relies on subordination and termination is validated by the programmer. To minimize the trusted core, programmers can avoid context subsumption by implementing context strengthening explicitly, but it comes with a substantial overhead. To further reduce the trusted core, Pientka et al (2014) have recently proposed a core calculus for primitive recursive functions and proven that it is normalizing. However, presently programs are not compiled to this core foundation.

Twelf provides a compact type checker to verify that relations manipulate well-formed derivations taking into account weakening. However, to validate that the relation describes a total function, we rely on mode, world and totlity checkers, which are outside of the LF framework and no certification exists.

Abella does not fit the LCF-style of proof assistants, nor does it provide a notion of proof certificate at this point; moreover, the annotation-based approach to (co)induction, as convenient as it is, introduces another level of indirection w.r.t. possible external checkers.

We end this section summing up some of the above discussion points in Table 1. The rows of the table summarize the following properties:

- The first two rows consider the representation of contexts both in OL specifications (OL ctx) and in the reasoning logic (RL ctx).
- The next two rows indicate how schemas are encoded (schemas) and how schema satisfaction is provided (schema sat).
- The following 4 rows indicate how we enforce structural properties (wk, str, exc) of contexts at the level of declarations (d-) and of whole contexts (c-).

- The next three rows describe how we ensure uniqueness of the elements in a context (uniq), and the substitution properties, both parametric (p subst) and hypothetical (h subst).
- Finally, the last two rows state the feasibility of the G and R versions.

In the table, LF stands for “Logical Framework” and CMTT stands for “Contextual Modal Type Theory,” the foundations of Twelf and Beluga, respectively. In both of these theories, Σ -types tie different declarations together.

9 Conclusion and Future Work

In this paper, we have formalized the benchmarks of the companion paper (Felty et al, 2014) in four systems: Twelf, Beluga, Hybrid, and Abella, and we have compared their general approach and solutions in some detail. In general, carrying out a qualitative comparison of a collection of very different systems is a daunting task, and this work represents a first step in this direction. This step involved focusing on one aspect, in particular reasoning about contexts, and finding enough common ground to be able to present proofs of the benchmarks in a way that highlights some of the similarities and differences of the various systems. This common ground included first identifying the difference between the R and G versions, whose initial idea can be seen in our early work (Felty and Pientka, 2010), as well as developing a theory of context of assumptions, identifying fine-grained structure of contexts along with their associated structural properties. This paper and the companion paper are very much intertwined. For example, our initial attempts at formalizing these benchmarks and comparing their proofs influenced the design of the theory of contexts, and vice versa, and the paper in its current form is the result of several such cycles of evolution.

As mentioned, our initial benchmarks focus on contexts, and thus expanding this set of benchmarks is an important direction for future work. A variety of ideas in this direction are discussed in the conclusion of the companion paper, and so we do not discuss it further here. Another direction of future work involves formalizing these benchmarks in other systems, both those that support HOAS, as well as those that take other approaches. We hope that others will implement these benchmarks and further our understanding of the similarities and differences, strengths and weaknesses of different systems and approaches. The future evolution of the ORBI specification language to better support this task is also discussed in the companion paper.

In addition to providing a means for comparison, benchmark problems also *prescribe* a kind of standard to which systems should adhere. In continuing the development of the systems considered here (as well as other systems that adopt these benchmarks), it will be important to ensure that proofs of these problems can still be done; if they cannot or if the proofs become more difficult, it will be important to explain and understand why.

With regard to the systems discussed in this paper, our survey has already sparked a number of developments to address the shortcomings in the various approaches and systems. For example, implementing the ∇ -quantifier is a current priority for Hybrid, and initial work on implementing a higher-order specification logic is underway. One of Abella’s development branch offers a plug-in (Savary-Belanger, 2013) that extends the system with a mechanism for declaring schemas

and contexts relations in a way reminiscent of Beluga; the aim is to automatically derive standard context results, such as member lemma stating inversion and uniqueness properties. Once this is in place, member lemmas can be in-lined and solved by calls to the plug-in tactics, significantly simplifying proof scripts.

Beluga is a system under active development and will continue to evolve. In the short term, we plan to support first-class substitutions also in the main release and improve upon source level syntax and reconstruction. Building on Pientka et al (2014), we plan to support interactive development of proofs in Beluga to facilitate writing proofs. This is an important step to improve and ease writing Beluga code. The recent support of first-class substitutions and context relations via indexed recursive datatypes and their use in mechanizing the completeness proof of algorithmic equality discussed in this paper also provides new insights into how proofs in proof-theoretic systems relate to Beluga programs; we see this as a first step towards not only comparing systems by examples, but starting to translate proofs done in one system to proofs in another.

Acknowledgements The first and third author acknowledge the support of the Natural Sciences and Engineering Research Council of Canada. We also thank Kaustuv Chaudhuri, Andrew Gacek, Nada Habli, Dale Miller, Gopalan Nadathur and Yuting Wang for discussing some aspects of the work with us. The first author would also like to extend her gratitude to the University of Ottawa's Women's Writers Retreats.

References

- Accattoli B (2012) Proof pearl: Abella formalization of λ -calculus cube property. In: Second International Conference on Certified Programs and Proofs, Springer, LNCS, vol 7679, pp 173–187
- Ambler SJ, Crole RL, Momigliano A (2003) A definitional approach to primitive recursion over higher order abstract syntax. In: ACM Workshop on MEchanized Reasoning about Languages with varIable biNding, ACM Press, pp 1–11
- Appel AW (2011) Verified software toolchain. In: Programming Languages and Systems, Springer, LNCS, vol 6602, pp 1–17
- Baelde D (2009) On the expressivity of minimal generic quantification. In: Third International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2008, Elsevier, ENTCS, vol 228, pp 3–19
- de Bruijn NG (1991) A plea for weaker frameworks. In: Huet G, Plotkin G (eds) Logical Frameworks, Cambridge University Press, pp 40–67
- Capretta V, Felty AP (2007) Combining de Bruijn indices and higher-order abstract syntax in Coq. In: Types for Proofs and Programs, International Workshop, TYPES 2006, Springer, LNCS, vol 4502, pp 63–77
- Cave A, Pientka B (2012) Programming with binders and indexed data-types. In: Thirty-Ninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, pp 413–424
- Cave A, Pientka B (2013) First-class substitutions in contextual type theory. In: Eighth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, ACM Press, pp 15–24
- Cave A, Pientka B (2014) Mechanizing logical relation proofs using contextual objects. Tech. rep., School of Computer Science, McGill

- Crary K (2009) Explicit contexts in LF (extended abstract). In: Third International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTTP 2008, ENTCS, vol 228, pp 53–68
- Dunfield J, Pientka B (2009) Case analysis of higher-order data. In: Third International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTTP 2008, Elsevier, ENTCS, vol 228, pp 69–84
- Felty A, Pientka B (2010) Reasoning with higher-order abstract syntax and contexts: A comparison. In: First International Conference on Interactive Theorem Proving, Springer, LNCS, vol 6172, pp 227–242
- Felty A, Momigliano A, Pientka B (2014) The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 1—a foundational view, submitted for publication simultaneously with this paper
- Felty AP (2002) Two-level meta-reasoning in Coq. In: International Conference on Theorem Proving in Higher-Order Logics, Springer, LNCS, vol 2410, pp 198–213
- Felty AP, Momigliano A (2009) Reasoning with hypothetical judgments and open terms in Hybrid. In: Eleventh International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, ACM Press, pp 83–92
- Felty AP, Momigliano A (2012) Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning* 48(1):43–105
- Gacek A (2008) The Abella interactive theorem prover (system description). In: Fourth International Joint Conference on Automated Reasoning, Springer, LNCS, vol 5195, pp 154–161
- Gacek A (2009) A framework for specifying, prototyping, and reasoning about computational systems. PhD thesis, University of Minnesota
- Gacek A, Miller D, Nadathur G (2011) Nominal abstraction. *Information and Computation* 209(1):48–73
- Gacek A, Miller D, Nadathur G (2012) A two-level logic approach to reasoning about computations. *Journal of Automated Reasoning* 49(2):241–273
- Habli N, Felty AP (2013) Translating higher-order specifications to Coq libraries supporting Hybrid proofs. In: Third International Workshop on Proof Exchange for Theorem Proving, EasyChair Proceedings in Computing, vol 14, pp 67–76
- Harper R, Licata DR (2007) Mechanizing metatheory in a logical framework. *Journal of Functional Programming* 17(4-5):613–673
- Harper R, Honsell F, Plotkin G (1993) A framework for defining logics. *Journal of the Association for Computing Machinery* 40(1):143–184
- Leroy X (2009) Formal verification of a realistic compiler. *CACM* 52(7):107–115
- McDowell RC, Miller DA (2002) Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic* 3(1):80–136
- Miller D, Nadathur G (2012) *Programming with Higher-Order Logic*. CUP
- Momigliano A (2012) A supposedly fun thing I may have to do again: A HOAS encoding of Howe’s method. In: Seventh ACM Workshop on Logical Frameworks and Meta-Languages, Theory and Practice, ACM Press, pp 33–42
- Momigliano A, Ambler SJ (2003) Multi-level meta-reasoning with higher order abstract syntax. In: Sixth International Conference on Foundations of Software Science and Computational Structures, Springer, LNCS, vol 2620, pp 375–391
- Momigliano A, Martin AJ, Felty AP (2008) Two-level Hybrid: A system for reasoning using higher-order abstract syntax. In: Second Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, ENTCS, vol 196, pp 85–93

- Nanevski A, Pfenning F, Pientka B (2008) Contextual modal type theory. *ACM Transactions on Computational Logic* 9(3):1–49, Article No. 23
- Pfenning F (2001) Computation and deduction, URL <http://www.cs.cmu.edu/~fp/courses/comp-ded/handouts/cd.pdf>, accessed 14 February 2014
- Pientka B (2005) Verifying termination and reduction properties about higher-order logic programs. *Journal of Automated Reasoning* 34(2):179–207
- Pientka B (2008) A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: *Thirty-Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, pp 371–382
- Pientka B (2010) Programming inductive proofs: A new approach based on contextual types. In: *Verification, Induction, Termination Analysis: Festschrift for Christoph Walther*, Springer, LNCS, vol 6463, pp 1–16
- Pientka B, Dunfield J (2008) Programming with proofs and explicit contexts. In: *Tenth International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, ACM Press, pp 163–173
- Pientka B, Dunfield J (2010) Beluga: A framework for programming and reasoning with deductive systems (system description). In: *Fifth International Joint Conference on Automated Reasoning*, Springer, LNCS, vol 6173, pp 15–21
- Pientka B, Ruan SS, Abel A (2014) Structural recursion over contextual objects. Tech. rep., School of Computer Science, McGill
- Rohwedder E, Pfenning F (1996) Mode and termination checking for higher-order logic programs. In: *Programming Languages and Systems: Sixth European Symposium on Programming*, Springer, LNCS, vol 1058, pp 296–310
- Savary-Belanger O (2013) Automated reasoning with contexts in Abella, URL <https://dl.dropboxusercontent.com/u/5648864/bilan.pdf>, code available from: <https://github.com/abella-prover/abella/tree/schemas>
- Schürmann C (2009) The Twelf proof assistant. In: *Theorem Proving in Higher Order Logics*, Springer, LNCS, vol 5674, pp 79–83
- Schürmann C, Pfenning F (1998) Automated theorem proving in a simple metalogic for LF. In: *Fifteenth International Conference on Automated Deduction*, Springer, LNCS, vol 1421, pp 286–300
- Schürmann C, Pfenning F (2003) A coverage checking algorithm for LF. In: *Sixteenth International Conference on Theorem Proving in Higher Order Logics*, Springer, LNCS, vol 2758, pp 120–135
- Tiu A, Momigliano A (2012) Cut elimination for a logic with induction and co-induction. *Journal of Applied Logic* 10(4):330–367
- Wang Y, Chaudhuri K, Gacek A, Nadathur G (2013) Reasoning about higher-order relational specifications. In: *Fifteenth International Symposium on Principles and Practice of Declarative Programming*, ACM Press, pp 157–168
- Zhao J, Nagarakatte S, Martin MMK, Zdancewic S (2012) Formalizing the LLVM intermediate representation for verified program transformations. In: *Symposium on Principles of Programming Languages*, ACM Press, pp 427–440