Abstract. The logical framework LF supports elegant encodings of formal systems using higher-order abstract syntax, modelling binders in the object language as binders in the metalanguage. However, reasoning about formal systems in LF via logical relations has been challenging. Implementing such proofs directly is beyond the logical strength of systems such as Twelf and Delphin. In this paper, we use the proof environment Beluga, which provides a dependently typed reasoning language on top of LF, to give a completeness proof of algorithmic equality. There are two key aspects of Beluga which we crucially rely upon: 1) we directly encode the logical relation using recursive types and higher-order functions 2) we exploit Beluga’s support for contexts and the equational theory of substitutions. This leads to a direct and compact mechanization, demonstrating Beluga’s strength at formalizing logical relations proofs.

Keywords: logical relations, logical frameworks, Beluga

1 Introduction

Proofs by logical relations play a fundamental role to establish rich properties such as contextual equivalence or normalization. This proof technique goes back to [Tait, 1967] and was later refined by Girard et al. [1990]. The central idea of logical relations is to specify relations on well-typed terms via structural induction on the syntax of types instead of directly on the syntax of terms themselves. Thus, for instance, logically related functions take logically related arguments to related results, while logically related pairs consist of components that are related pairwise.

Mechanizing logical relations proofs is challenging: first, specifying logical relations themselves typically requires a logic which allows arbitrary nesting of quantification and implications; second, to establish soundness of a logical relation, one must prove the Fundamental Property which says that any well-typed term under a closing simultaneous substitution is in the relation. This latter part requires some notion of simultaneous substitution together with the appropriate equational theory of composing substitutions. As Altenkirch [1993] remarked,

“I discovered that the core part of the proof (here proving lemmas about CR) is fairly straightforward and only requires a good understanding
of the paper version. However, in completing the proof I observed that in certain places I had to invest much more work than expected, e.g. proving lemmas about substitution and weakening.”

While logical normalization proofs often are not large, they are conceptually intricate and mechanizing them has become a challenging benchmark for proof environments. There are several key questions, when we attempt to formalize such proofs: How should we represent the abstract syntax tree for lambda-terms and enforce scope of bound variables? How should we represent well-typed terms or typing derivations? How should we deal with substitution? How can we define the logical relation on closed terms?

Early work by Altenkirch [1993], Berardi [1990], Coquand [1992] represented lambda-terms using (well-scoped) de Bruijn indices which leads to a substantial amount of overhead to prove properties about substitutions such as substitution lemmas and composition of substitution. To improve readability and generally better support such meta-theoretic reasoning, nominal approaches support α-renaming but substitution and properties about them are specified separately; the Isabelle Nominal package has been used in a variety of logical relations proofs from proving strong normalization for Moggi’s modal lambda-calculus [Doczkal and Schwinghammer, 2009] to mechanically verifying the meta-theory of LF itself including the completeness of equivalence checking [Narboux and Urban, 2008, Urban et al., 2011].

Approaches representing lambda-terms using higher-order abstract syntax (HOAS) trees (also called λ-tree syntax) model binders in the object language (i.e. in our case the simply typed lambda-calculus) as binders in the meta language (i.e. in our case the logical framework LF [Harper et al., 1993]). Such encodings inherit not only α-renaming and substitution from the meta-language, but also weakening and substitution lemmas. However, direct encodings of logical relations proofs is beyond the logical strength supported in systems such as Twelf [Pfenning and Schürmann, 1999]. In this paper, we demonstrate the power and elegance of logical relations proofs within the proof environment Beluga [Pientka and Dunfield, 2010] which is built on top of the logical framework LF. Beluga allows programmers to pair LF objects together with their surrounding context and this notion is internalized as a contextual type $\Psi.A$ which is inhabited by term $M$ of type $A$ in the context $\Psi$ [Nanevski et al., 2008]. Proofs about contexts and contextual LF objects are then implemented as dependently-typed recursive functions via pattern matching [Pientka, 2008, Pientka and Dunfield, 2008]. Beluga’s functional language supports higher-order functions and indexed recursive data-types [Cave and Pientka, 2012] which we use to encode the logical relation. As such it does not impose any restrictions as for example found in Twelf [Pfenning and Schürmann, 1999] which does not support arbitrary quantifier alternation or Delphin [Poswolsky and Schürmann, 2008] which lacks recursive data-types. Recently, Beluga has been extended to first-class simultaneous substitutions allowing abstraction over substitutions and supporting a rich equational theory about them [Cave and Pientka, 2013].
In this paper, we show that this infrastructure for modelling bound variables, contexts, open and closed objects, and simultaneous substitutions allow us to give a direct and compact mechanization of logical relation proofs. We illustrate its strength in this domain by considering the completeness proof of algorithmic equality for simply typed lambda-terms.

2 Proof Overview: Completeness of Algorithmic Equality

In this section we give a brief overview of the motivation and high level structure of the completeness proof of algorithmic equality. For more detail, we refer the reader to Crary [2005] and Harper and Pfenning [2005]. Extensions of this proof are important for the metatheory of dependently typed systems such as LF and varieties of Martin-Löf Type Theory, where they are used to establish decidability of typechecking. The proof concerns three judgements:

\[ \Gamma \vdash M \equiv N : A \] \quad \text{terms M and N are declaratively equivalent at type A}

\[ \Gamma \vdash M \leftrightarrow N : A \] \quad \text{terms M and N are algorithmically equivalent at type A}

\[ \Gamma \vdash M \leftrightarrow N : A \] \quad \text{paths M and N are algorithmically equivalent at type A}

Declarative equivalence (also called definitional equality) includes convenient but non-syntax directed rules such as transitivity and symmetry, among rules for congruence, extensionality and $\beta$-contraction. We will see the full definition in Sec. 3. In particular, it may include apparently type-directed rules such as extensionality at unit type:

\[ \Gamma \vdash M : \text{Unit} \quad \Gamma \vdash N : \text{Unit} \quad \Gamma \vdash M \equiv N : \text{Unit} \]

We do not consider such a rule in our development. For our present purpose it merely adds uninteresting cases to the proof. However, such a rule motivates the use of algorithmic type-directed equivalence. The rule above relies on type information and hence the usual normalize-and-compare strategy for deciding equivalence does not work in the presence of this rule. Algorithmic equivalence on the other hand is directly implementable, since it is directed by the syntax of types and terms. We define algorithmic term equivalence mutually with path equivalence, which is the syntactic equivalence of terms headed by variables, i.e. terms of the form $x \, M_1 \ldots M_n$.

In this paper, we show completeness of algorithmic equivalence for declarative equivalence. As usual, the problem is what to do at function types, since we need that applying equivalent functions to equivalent arguments yields equivalent results. This is the reason for defining an intermediate logical relation, which we write:

\[ \Gamma \vdash M \approx N : A \] \quad \text{terms M and N are logically equivalent at type A}

This relation is defined by induction on the structure of the type. The key case is at function type, where Crary defines:
\[ \Gamma \vdash M_1 \approx M_2 : A \Rightarrow B \quad \text{iff} \quad \begin{align*}
\text{for all } & \Delta \geq \Gamma \text{ and } N_1, N_2, \\
\text{if } & \Delta \vdash N_1 \approx N_2 : A \\
\text{then } & \Delta \vdash M_1 N_1 \approx M_2 N_2 : B
\end{align*} \]

The quantification over extensions \( \Delta \) of \( \Gamma \) is essential for establishing monotonicity, below. This Kripke-style monotonicity is one of the reasons that this proof is more challenging than normalization proofs, where this quantification can be avoided using other technical tricks. For our formalization, we take a slightly different approach which better exploits the features of Beluga available to us. We instead quantify over an arbitrary \( \Delta \) together with a renaming substitution \( \pi \) which brings terms from \( \Gamma \) to \( \Delta \), i.e. \( \Delta \vdash \pi : \Gamma \). This is a substitution of variables for variables, i.e. it embodies precisely the structural rules of exchange, contraction, and most importantly, weakening.

\[ \Gamma \vdash M_1 \approx M_2 : A \Rightarrow B \quad \text{iff} \quad \begin{align*}
\text{for all } & \Delta, N_1, N_2, \text{ and renamings } \pi \\
\text{if } & \Delta \vdash N_1 \approx N_2 : A \text{ and } \Delta \vdash \pi : \Gamma \\
\text{then } & \Delta \vdash M_1 [\pi] N_1 \approx M_2 [\pi] N_2 : B
\end{align*} \]

The high level goal is to establish that declaratively equivalent terms are logically equivalent, and that logically equivalent terms are algorithmically equivalent. The proof requires establishing a few key properties of logical equivalence. The first is monotonicity, which is crucially used for weakening logical equivalence. This is used when applying terms to fresh variables.

**Lemma 1 (Monotonicity).** If \( \Gamma \vdash M \approx N : A \) and \( \Delta \vdash \pi : \Gamma \) is a renaming substitution, then \( \Delta \vdash M[\pi] \approx N[\pi] : A \)

The second key property is (backward) closure of logical equivalence under weak head reduction. This is proved by induction on the type \( A \).

**Lemma 2 (Logical weak head closure).** If \( \Gamma \vdash N_1 \approx N_2 : A \) and \( M_1 \rightarrow^*_{wh} N_1 \) and \( M_2 \rightarrow^*_{wh} N_2 \) then \( \Gamma \vdash M_1 \approx M_2 : A \)

In order to escape logical equivalence to obtain algorithmic equivalence in the end, we need the main lemma, which is a mutually inductive proof showing that path equivalence is included in logical equivalence, and logical equivalence is included in algorithmic equivalence:

**Lemma 3 (Main lemma).**

1. If \( \Gamma \vdash M \leftrightarrow N : A \) then \( \Gamma \vdash M \approx N : A \)
2. If \( \Gamma \vdash M \approx N : A \) then \( \Gamma \vdash M \leftrightarrow N : A \)

Also required are symmetry and transitivity of logical equivalence, which in turn require symmetry and transitivity of algorithmic equivalence, determinacy of weak head reduction, and uniqueness of types. We will not go into detail about these lemmas, as they are relatively mundane, but refer the reader to the discussion in Crary [2005].

What remains is to show that declarative equivalence implies logical equivalence. This requires a standard technique to generalize the statement to all related instantiations of concrete terms:
Theorem 1 (Fundamental theorem). If $\Gamma \vdash M \equiv N : A$
and $\Delta \vdash \sigma_1 \approx \sigma_2 : \Gamma$ then $\Delta \vdash M[\sigma_1] \approx N[\sigma_2] : A$

By establishing the relatedness of the identity substitution to itself, i.e. $\Gamma \vdash \text{id} \approx \text{id} : \Gamma$ we can combine the fundamental theorem with the main lemma to obtain completeness.

Corollary 1 (Completeness). If $\Gamma \vdash M \equiv N : A$ then $\Gamma \vdash M \leftrightarrow N : A$

3 Mechanization

We mechanize the development of the declarative and algorithmic equivalence together with its completeness proof in Beluga, a dependently typed proof language built on top of the logical framework LF. The central idea is to specify lambda-terms, typing derivations and its small-step semantics in the logical framework LF. This allows us to model bindings uniformly using the LF function space and obviates the need to model and manage names explicitly. Beluga’s proof language allows programmers to encapsulate LF objects together with their surrounding context as contextual objects and provides support for higher-order functions, indexed recursive types, and pattern matching on contexts and contextual objects. We define type-directed declarative and algorithmic equivalence as well as the logical equivalence relation using indexed recursive types. All our proofs will then be implemented as recursive functions using pattern matching. The complete source code for our development can be found in the accompanying materials for the paper\textsuperscript{1}.

3.1 Encoding lambda-terms, typing and reduction in the logical framework LF

Our proof is about a simply-typed lambda calculus with one base type $i$. Extending the proof to support a unit type and products is straightforward. We describe the types and terms in LF as follows, employing HOAS for the representation of lambda abstraction. That is, we express the body of the lambda expression as an LF function $tm \rightarrow tm$. There is no explicit case for variables; they are implicitly handled by LF. We show side by side the corresponding grammar.

- $tp : type$
- $i : tp$
- $\Rightarrow : tp \rightarrow tp \rightarrow tp$. % infix

Types
- $T, S ::= i \mid T \Rightarrow S$

- $tm : type$
- $app : tm \rightarrow tm \rightarrow tm$
- $\text{lam} : (tm \rightarrow tm) \rightarrow tm$

Terms
- $M, N ::= x \mid \text{lam} \ x \ M \mid \text{app} \ M \ N$

Typing derivations are represented as a family of LF level types: we write oft $M T$ to say $M$ is of type $T$. In the lambda case, we quantify over an $x:tm$

\textsuperscript{1} Typechecking requires a recent branch of Beluga with support for first-class substitutions which is made available at http://comlogic.cs.mcgill.ca/beluga/.
together with an assumption oft x T that x of type T. This expresses “for all x, given oft x T we show oft (M x) S”. All free variables in the type declaration are implicitly universally quantified at the outside and type reconstruction will infer their type.

```
oft : tm → tp → type.
of/app : oft M (T ⇒ S) → oft N T → oft (app M N) S.
of/lam : ({x:tm} oft x T → oft (M x) S) → oft (lam M) (T ⇒ S).
```

Finally, we describe also weak head reduction for our terms. Notice here that the substitution of N into M in the β-reduction case is accomplished using LF application. We then describe multi-step reductions as a sequence of single step reductions.

```
step : tm → tm → type.
beta : step (app (lam M) N) (M N).
stepapp : step M M' → step (app M N) (app M' N).
mstep : tm → tm → type.
refl : mstep M M.
trans1 : step M M' → mstep M' M'' → mstep M M''.
```

### 3.2 Encoding declarative and algorithmic equivalence

Two forms of contexts are relevant for the proof. We describe these with schema definitions in Beluga. Schemas classify contexts in a similar way as LF types classify LF objects. Since contextual terms are always associated with a context in Beluga, we make use of a schema which lists only term variables, which we write ctx. We also use contexts which contain also typing derivations. These are grouped together with the tm variable they refer to, which we write block x:tm, d:oft x t.

```
schema ctx = tm;
schema tctx = some [t:tp] block x:tm, d:oft x t;
```

Although schemas are similar to Twelf’s world declarations, schema checking does not involve verifying that a given LF type family only introduces the assumptions specified in the schema; instead schemas will be used by the computation language to guarantee that we are manipulating contexts of a certain shape.

In this proof, we need the ability to relate our two notions of context. We employ a computation-level inductive datatype to express that a typing context is related to a term context. In the course of this proof, we employ the convention that Γ and Δ stand for typing contexts, while γ and δ stand for corresponding term contexts. In the base case, the empty contexts are related. In the step case, provided that Γ is related to γ, we say that Γ, b:block x:tm, d:oft x T[] is related to γ,x:tm – we simply drop the typing assumption².

² Our syntax for meta-variables such as T diverges slightly from present day Beluga. Presently, all metavariables must be associated with a substitution. If they are not, they are assumed to be closed and cannot depend on the context they occur in. In
data CtxRel : {Γ : tctx}{γ : ctx} ctype =
| RN1 : CtxRel .
| RCons : CtxRel Γ → CtxRel (Γ, b: block x: tm, d: oft x T[]) (γ, x: tm);

As for LF declarations, all free variables occurring in the recursive type are reconstituted and bound implicitly at the outside. We describe next the result of looking up the type of a variable \( x \) in \( γ \) in typing context \( Γ \) by its position. If \( x \) is the top variable of \( γ \), then its type in \( Γ \) is the type of the top variable of \( Γ \). Otherwise, if looking up the type of \( x \) in \( γ \) yields \( T \), then looking it up in an extended context also yields \( T \). Here we write \([γ, \; tm]\) for the contextual type of terms of type \( tm \) in context \( γ \), and \([\cdot, \; tp]\) for (closed) types. We use the letter \( ν \) for a metavariable standing for an object-level variable from \( γ \) (as opposed to a general term), and we associate \( ν \) with the \( ↑ \) weakening substitution to bring it from context \( γ \) to \( γ, x: tm \). We quantify over \((γ:ctx)\) in round parentheses, which indicates that it is implicit and recovered during reconstruction. Variables quantified in curly braces such as \( Γ\{γ:ctx\}\) are passed explicitly.

data Lookup : {Γ: tctx}(γ:ctx) \([γ, \; tm]\) → \([\cdot, \; tp]\) → ctype =
| Top : Lookup (Γ, b: block x: tm, y: oft x T[]) \([γ, x: tm, x]\) \([\cdot, \; T]\)
| Pop : Lookup Γ Γ → Lookup (Γ, b: block x: tm, y: oft x S[]) \([γ, x: tm, \nu]\) \([\cdot, \; T]\);

We now define declarative equality of terms, which includes non-algorithmic rules such as transitivity and symmetry. We note that it includes an extensionality rule, which states that for two terms \( M \) and \( N \) to be equal at arrow type, it suffices for them to be equal when applied to fresh variables, where freshness is achieved by weakening \( M \) and \( N \). Reflexivity on well-typed terms is omitted, as it adds non-essential complexity. It can instead be proved admissible, which we do not do here. For readability, we use a mixfix notation\(^3\). We write \([Γ \vdash \gamma, \; M \equiv \gamma, \; N : T]\) for declarative equivalence of \( M \) and \( N \) at type \( T \). Unbound metavariables such as \( M \) in \text{DecBeta} are implicitly universally quantified. They are passed implicitly and recovered during reconstruction.

data \(-≡-\) : {Γ: tctx}(γ:ctx) \([γ, \; tm]\) → \([γ, \; tm]\) → \([\cdot, \; tp]\) → ctype =
| DecBeta : Γ, b: block x: tm, d: oft x T[] \(\Vdash \gamma, x: tm, M2\) \equiv \([γ, x: N2 : \cdot, S]\)
  \(\rightarrow \; Γ \vdash \gamma, M1 : \gamma, N1 : \cdot, T\)
  \(\rightarrow \; Γ \vdash \gamma, \text{app}\; (\text{lam}\; (\text{\{id\}}))\; M1\equiv \gamma, \text{app}\; M1[N2[\cdot, \text{N1}]: \cdot, S]\)
| DecLam : Γ, b: block x: tm, d: oft x T[] \(\Vdash γ, x: tm, M\) \equiv \([γ, x: tm, N : \cdot, S]\)
  \(\rightarrow \; Γ \vdash \gamma, \text{lam}\; (\text{\{\{x\}\}})\; M\equiv \gamma, \text{app}\; \text{\{\{x\}\}}\; N : \cdot, \; T \Rightarrow S\)
| DecExt : Γ, b: block x: tm, d: oft x T[] \(\Vdash \gamma, x: tm, \text{app}\; M[\cdot, x]\) \equiv \([γ, \text{app}\; M[\cdot, x] : \cdot, S]\)
  \(\rightarrow \; Γ \vdash \gamma, N : \cdot, \nu : \cdot, T \Rightarrow S\)
| DecVar : Lookup Γ \([γ, \nu]\) \([\cdot, \; T]\)
  \(\rightarrow \; Γ \vdash \gamma, \nu \equiv \gamma, \nu : \cdot, \; T\)
| DecApp : Γ \(\vdash \gamma, M1\equiv \gamma, M2 : \cdot, \; T \Rightarrow S\)
  \(\rightarrow \; Γ \vdash \gamma, N1 \equiv \gamma, N2 : \cdot, \; T\)
| DecSym : Γ \(\vdash \gamma, M \equiv \gamma, N : \cdot, \; T\)
  \(\rightarrow \; Γ \vdash \gamma, \text{app}\; M\; M1\; N1 \equiv \gamma, \text{app}\; M2\; N2 : \cdot, \; S\)
| DecTrans : Γ \(\vdash \gamma, M \equiv \gamma, N : \cdot, \; T\)
  \(\rightarrow \; Γ \vdash \gamma, N \equiv \gamma, O : \cdot, \; T\)
  \(\rightarrow \; Γ \vdash \gamma, O \equiv \gamma, O : \cdot, \; T\);

\(^3\) Mixfix notation is not currently supported by Beluga.
We can now describe the algorithmic equality of terms. This is defined as two mutually inductive datatypes. We write \( \Gamma \vdash [\gamma, M] \Leftrightarrow [\gamma, N] : [\cdot, T] \) for algorithmic equivalence of terms \( M \) and \( N \) of type \( T \) in typing context \( \Gamma \). We write \( \Gamma \vdash [\gamma, M] \leftrightarrow [\gamma, N] : [\cdot, T] \) for algorithmic path equivalence – these are terms whose head is a variable, not a lambda abstraction. Term equality is directed by the type, while path equality is directed by the syntax. Two terms \( M \) and \( N \) at base type \( i \) are equivalent if they weak head reduce to weak head normal terms \( P \) and \( Q \) which are path equivalent. Two terms \( M \) and \( N \) are equivalent at type \( T \Rightarrow S \) if applying them to a fresh variable \( x \) of type \( T \) yields equivalent terms.

We remark that we must weaken the terms \( M \) and \( N \) from \( \gamma \) to \( \gamma, x : \text{tm} \) by associating them with the \( \uparrow \) weakening substitution. Variables are only path equivalent to themselves, and applications are path equivalent if the terms at function position are path equivalent, and the terms at argument positions are term equivalent.

\[
\text{data } _\downarrow \_ \downarrow _\cdot _\vdash _\cdot : \{\Gamma : \text{tctx}\}(\gamma : \text{ctx})\{\gamma, \text{tm}\} \rightarrow [\gamma, \text{tm}] \rightarrow [\cdot, \text{tp}] \rightarrow \text{ctx} =
\begin{align*}
&\text{AlgBase} : [\gamma, \text{mstep} M P] \rightarrow [\gamma, \text{mstep} N Q] \\
&\quad \rightarrow [\Gamma \vdash [\gamma, P] \leftrightarrow [\gamma, Q] : [\cdot, 1] \\
&\quad \rightarrow [\Gamma \vdash [\gamma, M] \leftrightarrow [\gamma, N] : [\cdot, 1] \\
&\text{AlgArr} : \Gamma, b : \text{block} x : \text{tm}, d : \text{oft} x T[\cdot] \vdash \\
&\quad [\gamma, x : \text{tm}, \text{app} M[\cdot] x] \leftrightarrow [\gamma, x : \text{tm}, \text{app} N[\cdot] x] : [\cdot, S] \\
&\quad \rightarrow [\Gamma \vdash [\gamma, M] \leftrightarrow [\gamma, N] : [\cdot, T \Rightarrow S]
\end{align*}
\]

and \( _\downarrow \_ \downarrow _\cdot : \{\Gamma : \text{tctx}\}(\gamma : \text{ctx})\{\gamma, \text{tm}\} \rightarrow [\gamma, \text{tm}] \rightarrow [\cdot, \text{tp}] \rightarrow \text{ctx} =
\begin{align*}
&\text{AlgApp} : \Gamma \vdash [\gamma, \text{m1}] \leftrightarrow [\gamma, \text{m2}] : [\cdot, T \Rightarrow S] \\
&\quad \rightarrow [\Gamma \vdash [\gamma, \text{m1}] \leftrightarrow [\gamma, \text{m2}] : [\cdot, T] \\
&\quad \rightarrow [\Gamma \vdash [\gamma, \text{app} \text{m1} \text{m1}] \leftrightarrow [\gamma, \text{app} \text{m2} \text{m2}] : [\cdot, S]]
\end{align*}

Below, we carve out a subset of all simultaneous substitutions – those which are well-typed variable-for-variable or renaming substitutions. These include compositions of the structural rules of exchange, weakening, and contraction only. We write \( \gamma[\delta] \) for the built-in type of simultaneous substitutions which take terms in \( \gamma \) to terms in \( \delta \). Beluga provides support for a rich equational theory around this built-in type, which we exploit heavily later.

\[
\text{data } _\downarrow \leftarrow _\cdot _\vdash _\cdot : \{\Delta : \text{tctx}\}(\delta : \text{ctx})\{\gamma : \text{ctx}\}{\sigma : \gamma[\delta]}\{\Gamma : \text{tctx}\} \text{ctx} =
\begin{align*}
&\text{Nil} : \Delta \vdash [\delta, \cdot] : \\
&\text{Cons} : \Delta \vdash [\delta, \sigma] : \Gamma \\
&\quad \rightarrow [\Delta \vdash [\delta, \sigma, \nu] : [(\Gamma, b : \text{block} x : \text{tm}, y : \text{oft} x T[\cdot])]]
\end{align*}
\]

We now show that algorithmic equivalence is monotonic (under renaming substitutions) where we omit some straightforward lemmas about renaming substitutions. Inductive proofs such as this are written recursively, where appeals to the induction hypotheses become recursive calls on structurally smaller terms.

\[
\text{rec } \text{algEqR_Monotone} : \Delta \vdash [\delta, \pi] : \Gamma \\
&\rightarrow \Gamma \vdash [\gamma, \text{m1}] \leftrightarrow [\gamma, \text{m2}] : [\cdot, \text{A}] \\
&\rightarrow \Delta \vdash [\delta, \text{m1}[\pi]] \leftrightarrow [\delta, \text{m2}[\pi]] : [\cdot, \text{A}]
\]

\[
\text{fn } \text{iv} \leftrightarrow \text{fn } \text{r} \leftrightarrow \text{case } \text{r} \text{ of } \\
\begin{align*}
&\text{AlgVar } \text{v} \leftrightarrow \text{AlgVar } (\text{vknVar } \text{v} \text{ iv}) \\
&\text{AlgApp } \text{r'} \text{ n'} \leftrightarrow \text{AlgApp } (\text{algEqR_Monotone iv } \text{r'}) (\text{algEqN_Monotone iv } \text{n'}))
\end{align*}
\]

8
and \( \text{algEqM.Monotone} : \Delta \vdash v[\delta, \pi] : \Gamma \rightarrow \Delta \vdash [\gamma. M1] \leftrightarrow [\gamma. M2] : [\mathcal{A}] \rightarrow \Delta \vdash [\delta. M1[\pi]] \leftrightarrow [\delta. M2[\pi]] : [\mathcal{A}] \)

\[\text{fn iv} \mapsto \text{let iv} : \Delta \vdash v[\delta, \pi] : \Gamma \rightarrow \Delta \vdash [\delta. M1[\pi]] \leftrightarrow [\delta. M2[\pi]] : [\mathcal{A}] \rightarrow \Delta \vdash v[\delta, \pi] : \Gamma \rightarrow \Delta \vdash [\delta. M1[\pi]] \leftrightarrow [\delta. M2[\pi]] : [\mathcal{A}] = \]

In the above proof, we remark that the apparently spurious let binding with a type annotation is a means to recover access to variables which were implicitly quantified (in this case, to recover \( \pi \)).

### 3.3 Encoding logical equivalence

Logical equality, written \( \Gamma \vdash [\gamma. M] \approx [\gamma. N] : [\mathcal{A}] \), expresses that \( M \) and \( N \) are logically related at type \( A \). We write this as a recursive type, which we note is well-defined because it is stratified on the type \( A \). At base type, two terms are logically related if they are algorithmically equivalent. At arrow type we employ a monotonicity condition: \( M1 \) is related to \( M2 \) in \( \Delta \) if, for any context \( \Delta \), renaming substitution \( \Delta \vdash \pi : \Gamma \), and \( N1 \), \( N2 \) related in \( \Delta \), we have that \( \text{app} M1[\pi] \approx \text{app} M2[\pi] \) in \( \Delta \). Traditionally, this monotonicity condition is expressed by quantifying over all extensions \( \Delta \) of \( \Gamma \). However, we can better exploit Beluga’s support for simultaneous substitution by quantifying over arbitrary contexts \( \Delta \) together with a renaming substitution \( \pi \) which allows us to pass from \( \Gamma \) to \( \Delta \). In the course of the development, \( \pi \) will only ever be instantiated with weakening substitutions, but this is not important for the proof.

\[
\begin{align*}
\text{data} \_\vdash \_\approx \_ : (\Gamma : \mathcal{C}x) \rightarrow (\gamma : \mathcal{C}tx) \rightarrow [\gamma. \text{tm}] \rightarrow [\gamma. \text{tm}] \rightarrow [\mathcal{T}] \rightarrow \mathcal{C}ype = \\
\text{LogBase} : \Gamma \vdash [\gamma. M] \leftrightarrow [\gamma. N] : [\mathcal{I}] \\
\rightarrow \Gamma \vdash [\gamma. M] \approx [\gamma. N] : [\mathcal{I}] \\
\text{LogArr} : \{M1 : [\gamma. \text{tm}] \} \rightarrow \{M2 : [\gamma. \text{tm}] \} \\
\rightarrow (\Delta : \mathcal{C}tx) \rightarrow \delta \rightarrow \pi : (\delta : \mathcal{C}tx) \rightarrow \{N1 : [\delta. \text{tm}] \} \rightarrow \{N2 : [\delta. \text{tm}] \} \\
\rightarrow \Delta \vdash v[\delta, \pi] : \Gamma \\
\rightarrow \Delta \vdash [\delta. N1] \approx [\delta. N2] : [\mathcal{T}] \\
\rightarrow \Delta \vdash v[N1[\pi]] \approx [\delta. \text{app} M1[\pi] N1] \approx [\delta. \text{app} M2[\pi] N2] : [\mathcal{S}] \\
\rightarrow \Gamma \vdash [\gamma. M1] \approx [\gamma. M2] : [\mathcal{T} \Rightarrow \mathcal{S}];
\end{align*}
\]

Crucially, logical equality is also monotonic under renaming substitutions. The proof proceeds in the arrow case by simply composing the two renaming substitutions and appealing to a lemma (not shown) which states that the composition of renaming substitutions is again a renaming. The proof is structurally recursive on the type, which is an implicit parameter. We use \( \lambda \) as the introduction form for universal quantifications over metavariables (contextual objects), for which we use uppercase and Greek letters, and \( \text{fn} \) with lowercase letters for computation-level function types (implications).

\[
\begin{align*}
\text{rec } \text{logEq.Monomotone} : \Delta \vdash v[\delta, \pi] : \Gamma \\
\rightarrow \Gamma \vdash [\gamma. M1] \approx [\gamma. M2] : [\mathcal{A}] \\
\rightarrow \Delta \vdash [\delta. M1[\pi]] \approx [\delta. M2[\pi]] : [\mathcal{A}] =
\end{align*}
\]
fn iv \rightarrow \text{let } iv : \Delta \vdash v [\delta. \pi] : \Gamma = iv \text{ in } fn e \rightarrow \text{case } e \text{ of }
| \text{LogBase } v \rightarrow \text{LogBase } (\text{algEqN-Monotone } iv \ v)
| \text{LogArr } [\gamma. \ M1] [\gamma. \ M2] f \rightarrow 
\text{LogArr } [\delta. \ M1[\pi]] [\delta. \ M2[\pi]]
\quad (\lambda \Delta' \delta' \pi. \pi[\pi]) [\delta'. \ M1[\delta'. \ N2] \text{ rh } (\text{renCompose } iv \ iv') \text{ rh })

The main lemma is mutually recursive, expressing that path equivalence is included in logical equivalence, and logical equivalence is included in algorithmic term equivalence. This enables “escaping” from the logical relation to obtain an algorithmic equality in the end. They are structurally recursive on the type, which we leave as an implicit parameter to $\text{reify}$. Crucially, in the arrow case, $\text{reify}$ instantiates the higher order parameter $f$ with a weakening substitution and the corresponding fresh variable.

We show here that logical equivalence is backward closed under weak head reduction, appealing to some unshown minor lemmas about $m\text{step}$ such as transitivity. Again this is structurally recursive on the type, which is implicit. At base type, we simply appeal to transitivity of $m\text{step}$. At arrow type, we must reduce under the function position of application. Underscores are used where values are inferred by reconstruction.

3.4 Fundamental lemma

The fundamental theorem requires us to speak of all instantiations of open terms by related substitutions. We express here the notion of related substitutions.
Trivially, empty substitutions are related at empty domain. If \( \sigma_1 \) and \( \sigma_2 \) are related at \( \Gamma \) and \( M_1 \) and \( M_2 \) are related at \( T \), then \( \sigma_1, M_1 \) and \( \sigma_2, M_2 \) are related at \( \Gamma, b : \text{block} \ x : \text{tm}, d : \text{oft} \ x \ T[1] \).

The fundamental theorem requires a proof that \( M_1 \) and \( M_2 \) are declaratively equal, together with logically related substitutions \( \sigma_1 \) and \( \sigma_2 \), and produces a proof that \( M_1[\sigma_1] \) and \( M_2[\sigma_2] \) are logically related. A key case of interest is the \texttt{lam} case. The complete proof is in Fig. 1. We remark that we do not perform any equational reasoning about substitutions, as this is handled implicitly by Beluga’s support for simultaneous substitutions. In the transitivity and symmetry cases, we appeal to transitivity and symmetry of logical equivalence, which we omit. In turn, these require transitivity and symmetry of algorithmic equivalence, and a handful of lemmas about determinacy of evaluation and typing.

We describe a few cases of the proof in detail, in particular the application and lambda abstraction cases. In the application case, we have the following:

1. \( d_1 : \Gamma \vdash [\gamma. M_{11}] \equiv [\gamma. M_{12}] : [\cdot. T \Rightarrow S] \)
2. \( d_2 : \Gamma \vdash [\gamma. M_{21}] \equiv [\gamma. M_{22}] : [. T] \)
3. \( r : \text{CtxRel} \triangleright \delta \)
4. \( s : \Delta \vdash [\delta, \sigma_1] \equiv [\delta, \sigma_2] : \Gamma \)
5. \( \text{thm} \ d_1 \ r \ s : \Delta \vdash [\delta, M_{11}[\sigma_1]] \equiv [\delta, M_{12}[\sigma_2]] : [. T \Rightarrow S] \) (by I.H.)
6. \( \text{thm} \ d_2 \ r \ s : \Delta \vdash [\delta, M_{21}[\sigma_1]] \equiv [\delta, M_{22}[\sigma_2]] : [. T] \) (by I.H.)
7. \( f : \{\Delta' : \text{ctx}\} \{\delta' : \text{ctx}\} \{\pi : \delta'\} \{(N_1 : [\delta'. \text{tm}]) \{(N_2 : [\delta'. \text{tm}])\} \}
   \quad \text{CtxRel} \Delta' \delta'
   \quad \rightarrow \Delta' \vdash v [\delta', \pi] : \Delta
   \quad \rightarrow \Delta' \vdash [\delta', N_1] \approx [\delta', N_2] : [. T]
   \quad \rightarrow \Delta' \vdash [\delta', \text{app} M_{11}[\sigma_1][\pi], N_1] \approx [\delta', \text{app} M_{12}[\sigma_2][\pi], N_2] : [. S]
   \) (from inversion using LogArr constructor, line 5)
8. \( \text{idIsRen} \ r : \Delta \vdash v [\delta, \text{id}] : \Delta \)
9. \( f \Delta \delta [\delta, \text{id}] [\delta, M_{21}[\sigma_1]] [\delta, M_{22}[\sigma_2]] \ r \ (\text{idIsRen} \ r) \ (\text{thm} \ d_2 \ r \ s)
   : \Delta \vdash [\delta, \text{app} M_{11}[\sigma_1][\text{id}], M_{21}[\sigma_1]]
   \equiv [\delta, \text{app} M_{12}[\sigma_2][\text{id}], M_{22}[\sigma_2]] : [. S]
   \) (by applying line 7 to line 6)
10. \( f \Delta \delta [\delta, \text{id}] [\delta, M_{21}[\sigma_1]] [\delta, M_{22}[\sigma_2]] \ r \ (\text{idIsRen} \ r) \ (\text{thm} \ d_2 \ r \ s)
   : \Delta \vdash [\delta, \text{app} M_{11}[\sigma_1], M_{21}[\sigma_1]] \approx [\delta, \text{app} M_{12}[\sigma_2], M_{22}[\sigma_2]] : [. S]
    \) (by equational theory of substitutions)

As required. The last step requires the identity property of composition of substitutions, which is built into Beluga's notion of equality for contextual objects. These steps are composed into the two lines for the DecApp case in Fig. 1.

The lambda abstraction case is somewhat more involved and proceeds as follows:

1. \( d_1 : \Gamma, b : \text{block} \ x : \text{tm}, d : \text{of} \ t \{[\gamma, x : \text{tm} \ M_1] \equiv [\gamma, x : \text{tm} \ M_2] : [. S] \)
2. \( r : \text{CtxRel} \triangleright \delta \)
3. \( s : \Delta \vdash [\delta, \sigma_1] \equiv [\delta, \sigma_2] : \Gamma \)
4. Suppose we are given \( \Delta' : \text{ctx}, \delta' : \text{ctx}, \pi : \delta' \), \( N_1 : [\delta'. \text{tm}], N_2 : [\delta'. \text{tm}] \),
   \( r_h : \text{CtxRel} \Delta' \delta', \text{iv} : \Delta' \vdash v \pi : \Delta, \)
   \( r_n : \Delta' \vdash [\delta', N_1] \approx [\delta', N_2] : [. T] \)
5. \( \text{Cons} (\text{wknLogEqSub} \ iv \ s) \ r_n : \Delta' \vdash s [\delta', \sigma_1[\pi], N_1] \approx [\delta', \sigma_2[\pi], N_2] : \Gamma \)
   \) (by construction)
6. \( q_2 : \Delta' \vdash [\delta', M_1[\sigma_1[\pi]], N_1] \approx [\delta', M_2[\sigma_2[\pi]], N_2] : [. S] \)
   \) (by I.H. with line 1, line 5)
7. \( q_2 : \Delta' \vdash [\delta', M_1[\sigma_1][\downarrow], x][\text{id}_\delta, N_1] \approx [\delta', M_2[\sigma_2[\pi]][\downarrow], x][\text{id}_\delta, N_2] : [. S] \)
   \) (by equational theory of substitutions)
8. \( \Delta' \vdash [\delta', \text{app} (\lambda \, x. M_1[\sigma_1[\downarrow]], x)) N_1] \approx [\delta', \text{app} (\lambda \, x. M_2[\sigma_2[\pi]][\downarrow], x)) N_2] : [. S] \)
   \) (by lemma closed under \( \beta \))
9. \( \Delta' \vdash [\delta', \text{app} (\lambda \, x. M_1[\sigma_1[\downarrow], x)] N_1] \approx [\delta', \text{app} (\lambda \, x. M_2[\sigma_2[\pi]], x)] N_2] : [. S] \)
   \) (by equational theory of substitutions)
10. \( \Delta \vdash [\delta, \text{lam} \ (\lambda \, x. \text{M}_1[\sigma_1[\downarrow], x])] \equiv [\delta, \text{lam} \ (\lambda \, x. \text{M}_2[\sigma_2[\downarrow], x])] : [. S] \)
   \) (by LogArr definition)
This case clearly relies heavily on the equational theory of substitutions, which is built into Beluga’s notion of equality on contextual objects. This means that these equational steps do not appear in the formal proof, seen in Fig. 1 as the DecLam case. Completeness is an easy corollary of the fundamental theorem, instantiating the substitutions to be identities and appealing to the main lemma to escape the logical relation.

\[
\text{rec idLogEqSub : CtxRel Γ γ} \rightarrow \Gamma \vdash s [\gamma. \text{id}] \approx [\gamma. \text{id}] : \Gamma = \ldots
\]

\[
\text{rec completeness : CtxRel Γ γ} \rightarrow \Gamma \vdash [\gamma. \text{M1}] \equiv [\gamma. \text{M2}] : [\cdot. \text{T}]
\]

\[
\rightarrow \Gamma \vdash [\gamma. \text{M1}] \leftrightarrow [\gamma. \text{M2}] : [\cdot. \text{T}] =
\]

\[
\text{fn r} \mapsto \text{fn e} \mapsto \text{reify r (thm e r (idLogEqSub r))};
\]

Establishing soundness and decidability of declarative equality is then straightforward. We remark that the completeness theorem can in fact be executed, viewing it as an algorithm for normalizing derivations in the declarative system to derivations in the algorithmic system. The extension to a proof of decidability would be a correct-by-construction functional algorithm for the decision problem. This is a unique feature of carrying out the proof in a type-theoretic setting like Beluga, where the proof language also serves as a computation language. In total, we manually proved only 6 (trivial) lemmas about substitutions which we would consider bureaucratic. These were constructions on renaming substitutions, and we refer the reader to the accompanying source code for details.

The proof passes Beluga’s typechecking and coverage checking, but we must add the disclaimer that Beluga does not presently perform termination checking or positivity analysis for inductive datatypes. It is however relatively straightforward for the reader to convince himself that all of our recursive functions are structurally recursive, either on types or on inductively defined derivations, and hence terminating. We note that the definition of the logical relation is of course not positive due to the recursive occurrence on the left of an arrow, so we have been careful to call it a recursive type as opposed to an inductive type. We argue for its well-definedness by a different criteria, namely that it is stratified by its type index. Lemmas which operate on it are then typically structurally recursive on the type. In the future we plan to make this analysis precise and distinguish between inductive types and stratified recursive types in the theory and implementation. Another approach is to justify the definition of the logical relation by large elimination – the computation of a type by recursion on a value. This approach requires more drastic extensions to the metatheory of Beluga.

4 Related Work

Mechanizing proofs by logical relations is an excellent benchmark to evaluate the power and elegance of a given proof development. Because it requires nested quantification and recursive definitions, encoding logical relations has been particularly challenging for systems supporting HOAS encodings.

There are two main approaches to support reasoning about HOAS encodings: 1) In the proof-theoretic approaches, we adopt a two-level system where we im-
plement a specification logic (similar to LF) inside a higher-order logic supporting (co)inductive definitions, the approach taken in Abella [Gacek, 2008], or type theory, the approach taken in Hybrid [Momigliano et al., 2008]. To distinguish in the proof theory between quantification over variables and quantification over terms, Gacek et al. [2008] introduce a new quantifier, $\nabla$, to describe nominal abstraction logically. To encode logical relations one uses recursive definitions which are part of the reasoning logic [Gacek et al., 2009]. Induction in these systems is typically supported by reasoning about the height of a proof tree; this reduces reasoning to induction over natural numbers, although much of this complexity can be hidden in Abella. Compared to our development in Beluga, Abella lacks support for modelling a context of assumptions and simultaneous substitutions. As a consequence, some of the tedious basic infrastructure to reason about open and closed terms and substitutions still needs to be built and maintained. Moreover, Abella’s inductive proofs cannot be executed and do not yield a program for normalizing derivation.

2) The type-theoretic approaches fall into two categories: we either remain within the logical framework and encode proofs as relations as advocated in Twelf [Pfenning and Schürmann, 1999] or we build a dependently typed functional language on top of LF to support reasoning about LF specifications as done in Beluga. The former approach lacks logical strength; the function space in LF is “weak” and only represents binding structures instead of computations. To circumvent these limitations, Schürmann and Sarnat [2008] proposes to implement a reasoning logic within LF and then use it to encode logical relation arguments. This approach scales to richer calculi [Rasmussen and Filinski, 2013] and avoids reasoning about contexts, open terms and simultaneous substitutions explicitly. However, one might argue that it not only requires additional work to build up a reasoning logic within LF and prove its consistency, but is also conceptually different from what one is used to from on-paper proofs. It is also less clear whether the approach scales easily to proving completeness of algorithmic equality due to the need to talk about context extensions in the definition of logical equivalence of terms of function type.

Outside the world of HOAS, Narboux and Urban [2008] have carried out essentially the same proof in Nominal Isabelle, and later Urban et al. [2011] tackle the extension from the simply-typed lambda calculus to LF. Relative to their approach, Beluga gains substitution for free, but more importantly, equations on substitutions are silently discharged by Beluga’s built-in support for their equational theory, so they do not even appear in proofs. In contrast, proving these equations manually requires roughly a dozen intricate lemmas.

5 Conclusion

Our implementation of completeness of algorithmic equality takes advantage of key infrastructure provided by Beluga: it utilizes first-class substitutions, contexts, contextual objects and the power of recursive types. This yields a direct and compact implementation of all the necessary proofs which directly corre-
spond to their on-paper developments. Moreover, our proof yields an executable program. While more work on Beluga’s frontend will improve and make simpler such developments, we have demonstrated that the core language is not only suitable for standard structural induction proofs such as type safety, but also proofs by logical relations.

References


