Under consideration for publication in Math. Struct. in Comp. Science

Mechanizing Proofs with Logical Relations – Kripke-style

Andrew Cave and Brigitte Pientka

School of Computer Science, McGill University, Montreal, Canada

Received 15 August 2017

Proofs with logical relations play a key role to establish rich properties such as normalization or contextual equivalence. They are also challenging to mechanize. In this paper, we describe two case studies using the proof environment Beluga: first, we explain the mechanization of the weak normalization proof for the simply-typed lambda-calculus; second, we outline how to mechanize the completeness proof of algorithmic equality for simply typed lambda-terms where we reason about logically equivalent terms. The development of these proofs in Beluga relies on three key ingredients: 1) we encode lambda-terms together with their typing rules, operational semantics, algorithmic and declarative equality using higher-order abstract syntax thereby avoiding the need to manipulate and deal with binders, renaming and substitution ourselves. 2) we take advantage of Beluga's support for representing derivations that depend on assumptions and first-class contexts to directly state inductive properties such as logical relations and inductive proofs 3) we exploit Beluga's rich equational theory for simultaneous substitutions; as a consequence users do not need to establish and subsequently use substitution properties, and proofs are not cluttered with references to them. We believe these examples demonstrate that Beluga provides the right level of abstractions and primitives to mechanize challenging proofs using higher-order abstract syntax encodings. They also demonstrate how engaging and following Beluga's framework of thinking about contextual objects, contexts, and context extensions sharpens our mathematical thought processes providing a uniform, Kripke-style perspective of logical relations proofs.

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 [Tai67] and was later refined by Girard [GLT90]. 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 [Alt93] 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 that are highlighted when we attempt to formalize such proofs: Are the terms we are reasoning about closed? If they are not closed, how can we characterize their free variables and reason about them? How should we represent the abstract syntax tree for lambda-terms and enforce the scope of bound variables? How should we represent well-typed terms or typing derivations? How should we deal with substitution? How can we define logical relations on open terms?

Early work [Ber90, Coq92, Alt93] 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 [DS09] to mechanically verifying the meta-theory of LF itself including the completeness of equivalence checking [NU08, UCB11].

Approaches representing lambda-terms using higher-order abstract syntax (HOAS) 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 [HHP93]). 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 are beyond the logical strength supported in systems such as Twelf [PS99]. In this paper, we demonstrate the elegance of logical relations proofs within the proof environment Beluga [PD10] which is built on top of the contextual logical framework LF. In contrast to LF where the assumptions are implicit, the contextual logical framework LF extends LF with first-class contexts and first-class simultaneous substitutions supporting a rich equational theory about them [CP13,PC15]. Moreover, it allows the direct representation of derivations that depend on assumptions by pairing LF objects together with their surrounding context and this notion is internalized as a contextual type $[\Psi \vdash A]$ which is inhabited by term M of type A in the context Ψ [NPP08].

Properties about contexts, substitutions, and contextual objects such as for example logical relations can be encoded in Beluga using indexed inductive types [CP12]. Inductive proofs about contexts and contextual LF objects are implemented in Beluga as dependently-typed recursive functions via pattern matching [Pie08, PD08].

In this paper, we describe two case-studies, both of which concentrate on the simply-

typed lambda-calculus and require reasoning about open terms: first, we explain the mechanization of the weak normalization proof for the simply-typed lambda-calculus using logical relations. As we reduce under lambda-abstractions, we must define reducibility on open terms. This is an example of a unary logical relation. Second, we outline how to mechanize the completeness proof of algorithmic equality for simply typed lambda-terms by Crary [Cra05]. Algorithmic equality is defined in a type-directed way and two terms M and N of type $A \to B$ are equal, if for any term x, $M\ x$ is equal to $N\ x$. To prove completeness we reason about logically equivalent open terms using a binary logical relation. In these case-studies, we rely upon three key aspects:

1 We encode lambda-terms together with their typing rules, operational semantics, algorithmic and declarative equality using higher-order abstract syntax in the logical framework LF. This allows us to model binders in our object language (i.e. the simply-typed lambda-calculus) using the binders in the logical framework LF. As a consequence, we do not need to build up our own infrastructure for representing binders, renaming and substitution. We showcase two techniques of how to represent and reason with well-typed terms: in our first case study of weak normalization we will work with intrinsically typed terms, while in our second case study of proving completeness of algorithmic equality we reason about well-typed terms using typing derivations explicitly.

2 We give a Kripke-style definition of logical relations about terms. Kripke logical relations are indexed by possible worlds and reason about arbitrary future extensions of worlds. In our setting we define the logical relation on terms M in a context Ψ in which it is defined. The context hence plays the role of the world in which the term is meaningful. Possible context extensions are characterized by simultaneous substitutions. This allows us to argue that logically related terms in a context Ψ and are also logically related in extensions of the context Ψ. Letting ourselves be guided by the underlying philosophy of the contextual logical framework LF sharpens our mathematical thought processes leading to a generalized and uniform Kripke-style formulation of logical relations on open terms.

3 We take advantage of Beluga's support for representing derivations that depend on assumptions using contextual objects, first-class contexts, and first-class simultaneous substitutions to directly state and encode inductive properties such as logical relations and proofs about well-typed terms. In particular, we exploit these features to give a Kripke-style definition of reducibility candidates for terms together with the context in which they are meaningful using inductive types and higher-order functions. In mechanizing the proofs we benefit from Beluga's rich equational theory of simultaneous substitutions which obliterates the need to establish various properties about simultaneous substitution and avoids cluttering our proofs with them. This leads to a direct, elegant, and compact mechanization and allows us to demonstrate Beluga's strength at formalizing logical relations proofs.

All case studies described in this paper, in addition to various variations, are accessible at `https://github.com/Beluga-lang/Beluga/tree/master/examples/logrel`.

2. Example: Weak Normalization

We begin discussing the weak normalization proof for the simply typed lambda-terms with reduction under binders.

2.1. *The Basics: Well-typed Lambda-Terms*

The grammar for our lambda-calculus is straightforward: it includes lambda abstractions, variables, and application in addition to simple types formed by the base type $\mathbf{i}$ and function types.

$$\begin{array}{lrcl} \text{Types} & A, B & ::= & \mathbf{i} \mid A \to B \\ \text{Terms} & M, N & ::= & x \mid \mathsf{lam}\ x.M \mid M\ N \\ \text{Typing Context} & \Gamma, \Psi & ::= & \cdot \mid \Gamma, x : A \end{array}$$

We choose to describe the typing rules, the operational semantics, and normal forms in a two-dimensional way. Following Gentzen's natural deduction style we keep the context of assumptions implicit. There are two main reasons for this kind of presentation: first, it can be directly translated into specifications in the logical framework LF and hence it foreshadows our mechanization; second, letting us be guided by LF highlights clearly what assumptions are made and will naturally lead us to a concrete characterization of contexts which we postpone until needed in the next section. We begin stating the rules for well-typed terms.

$\boxed{M : A}$ Term M has type A

$$\frac{M : A \to B \quad N : A}{M\ N : B}\ \texttt{app} \qquad \frac{\begin{array}{c}\overline{x : A}^{\,u} \\ \vdots \\ M : B\end{array}}{\mathsf{lam}\ x.M : A \to B}\ \texttt{lam}^{x,u}$$

Note that in the typing rule for lambda-abstractions, we must show that M has type B assuming that x has type A and x is a fresh variable. We then define a form of reduction, written as $M \longrightarrow N$, which supports reductions in the body of a lambda-abstraction. On top of single step reduction, we build a multi-step relation, written as $M \longrightarrow^* N$, which includes transitivity and reflexivity.

$\boxed{M \longrightarrow^* N}$ M steps to N in multiple steps

$$\frac{}{M \longrightarrow^* M}\ \texttt{s\_refl} \qquad \frac{M \longrightarrow M' \quad M' \longrightarrow^* N}{M \longrightarrow^* N}\ \texttt{s\_trans}$$

$\boxed{M \longrightarrow N}$ M steps to N in one step

$$\frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N}\ \texttt{s\_app1} \qquad \frac{N \longrightarrow N'}{M\ N \longrightarrow M\ N'}\ \texttt{s\_app2} \qquad \frac{M \longrightarrow M'}{\mathsf{lam}\ x.M \longrightarrow \mathsf{lam}\ x.M'}\ \texttt{s\_lam}^{x}$$

$$\frac{}{(\mathsf{lam}\ x.M)\ N \longrightarrow [N/x]M}\ \texttt{s\_beta}$$

We halt when we reach a normal form, i.e. the term does not contain a redex. We define below when a term is in normal form in a judgmental way using two mutual judgments.

$\boxed{M\ \mathsf{norm}}\ M$ is normal $\qquad$ $\boxed{M\ \mathsf{neut}}\ M$ is neutral $\qquad$ $\dfrac{\dfrac{}{x\ \mathsf{neut}}\ n}{\begin{array}{c}\vdots\\ M\ \mathsf{norm}\end{array}}\ \mathtt{n\_lam}^{x,n}$

$$\frac{M\ \mathsf{neut}}{M\ \mathsf{norm}}\ \mathtt{n\_neut} \qquad \frac{M\ \mathsf{neut} \quad N\ \mathsf{norm}}{M\ N\ \mathsf{neut}}\ \mathtt{n\_app}$$

2.2. *Hand-written Proof Outline: Weak Normalization*

We revisit here the proof that the evaluation of well-typed terms halts using reducibility candidates which is typical for logical relations proofs. However, often we simply define reducibility on closed terms. As we allow reductions under binders, we must state reducibility on open terms. This naturally leads to the question: what are the free variables in a term? How can we characterize them and reason about them? – Here we propose to think of a term within a context of assumptions and state reducibility of a term at type A using Kripke-style possible worlds where the context in which a term is meaningful corresponds to a world.

As is standard, reducibility is defined inductively on the type. A well-typed term is reducible at base type precisely when it halts. Recall that a well-typed term M halts, if M is in normal form; however, our judgment $M\ \mathsf{norm}$ requires that for each variable occurring in M, we have an assumption stating that it is neutral. When we state properties about well-typed terms such as reducibility, it is often more convenient and more precise to make the context of assumptions explicit. This is in contrast to our previous two-dimensional representation which left the context of assumptions implicit. Obviously, the two formulations are equivalent and it is easy to convert between them and we choose here the representation that is most convenient and most precise at a given point.

This leads us to defining the context in which a term M is meaningful as a typed neutral variable context below where we keep track of typing assumptions as well as the fact that all variables are neutral .

$$\text{Typed Neutral Variable Context} \quad \Psi, \Phi \quad ::= \quad \cdot \mid \Psi, x : A, x\ \mathsf{neut}$$

Hence, we define more precisely that a term M in the context Ψ (written as $\Psi \vdash M$) is reducible at base type, when it halts, i.e. there exists a term V it steps to and $V\ \mathsf{norm}$ in the context Ψ.

How can we now define reducibility for a term M at function type $A \to B$? – In the course of evaluating a term M, we may extend the initial world (i.e. the context Ψ in which M is meaningful) as we traverse lambda-abstractions. We hence must be able to reason about future extensions of a world (i.e. contexts Φ) and more importantly, we must be able to use the term M in the future world. The notion of accessibility of worlds in Kripke-style semantics corresponds in our setting to moving from one context Ψ to its extension via a simultaneous substitution that weakens Ψ and guarantees that normal forms are preserved, i.e. we only replace variables with neutral terms. In practice, we in fact simply rename variables. We can then define reducibility of a term M in a context Ψ at function type $A \to B$, if for all future contexts Φ, neutral substitutions ρ,

where $\Phi \vdash \rho : \Psi$, and for every reducible term N at type A in the future context Φ, the application $[\rho]M\ N$ is reducible at type B in the future context Φ.

We define first the abbreviation $\Phi \geq_\rho \Psi$ to mean $\Phi \vdash \rho : \Psi$. The substitution ρ is neutral, i.e. it only allows variables in Ψ to be replaced by neutral terms of the expected type.

$$\boxed{\Phi \geq_\rho \Psi}\ \rho \text{ is a neutral substitution from the context } \Psi \text{ to } \Phi$$

$$\frac{}{\Phi \geq_{\cdot} \cdot} \qquad \frac{\Phi \geq_\rho \Psi \quad \Phi \vdash M : A \quad \Phi \vdash M\ \mathsf{neut}}{\Phi \geq_{\rho, M/x} \Psi, x : A, x\ \mathsf{neut}}$$

This leads to the following definition of reducibility on open terms.

$$\begin{array}{lcl} \mathcal{R}_{\mathbf{i}} & = & \{\Psi \vdash M \mid \Psi \vdash M \text{ halts}\} \\ \mathcal{R}_{A\to B} & = & \{\Psi \vdash M \mid \ \Psi \vdash M \text{ halts and } \forall \Phi \geq_\rho \Psi, N \text{ where } \Phi \vdash N : A, \\ & & \qquad \text{if } (\Phi \vdash N) \in \mathcal{R}_A \text{ then } (\Phi \vdash [\rho]M\ N) \in \mathcal{R}_B\} \end{array}$$

Standard definitions of reducibility often omit keeping and maintaining typing information. We choose to make it explicit. Our mechanization will implicitly keep track of typing information using intrinsically typed terms.

The proof for showing that all well-typed terms halt, falls into two main parts. The first part, the main lemma (sometimes called the escape lemma), states that if a term is reducible at type A then it halts. The fundamental theorem states that all well-typed terms are indeed reducible. Both theorems together then allow us to show that well-typed terms halt. In stating the theorems we make for clarity the context explicit in our typing judgments and when referring to our judgmental definition of normal and neutral terms.

Theorem 2.1 (Main lemma).

1 If $(\Psi \vdash M) \in \mathcal{R}_A$ then $\Psi \vdash M$ halts.
2 If $\Psi \vdash M : A$ and $\Psi \vdash M\ \mathsf{neut}$ then $(\Psi \vdash M) \in \mathcal{R}_A$.

Proof. The first part follows directly from the definition of reducibility. The second part is proven by induction on the type A. □

One can then easily prove that $\mathcal{R}_A$ is closed under expansion.

Lemma 2.2 (Closure under expansion).

1 If $(\Psi \vdash M') \in \mathcal{R}_A$ and $M \longrightarrow M'$ then $(\Psi \vdash M) \in \mathcal{R}_A$.
2 If $(\Psi \vdash M') \in \mathcal{R}_A$ and $M \longrightarrow^* M'$ then $(\Psi \vdash M) \in \mathcal{R}_A$.

Proof. Both statements are proven by induction on A. We show here the proof of the first statement.

Case: $A = \mathbf{i}$

$(\Psi \vdash M') \in \mathcal{R}_{\mathbf{i}}$	by assumption
$\Psi \vdash M'$ halts	by definition of $\mathcal{R}_{\mathbf{i}}$
$\exists V.\ M' \longrightarrow^* V$ and $\Psi \vdash V\ \mathsf{norm}$	by definition of halts
$M \longrightarrow^* V$	by rule `s_trans`

$\Psi \vdash M$ halts — by definition of halts
$(\Psi \vdash M) \in \mathcal{R}_{\mathbf{i}}$ — by definition of $\mathcal{R}_{\mathbf{i}}$

Case: $A = B \to C$
$(\Psi \vdash M') \in \mathcal{R}_{B \to C}$ — by assumption
$(\Psi \vdash M')$ halts, i.e. $\exists V. M' \longrightarrow^* V$ and $\Psi \vdash V$ norm
for all $\Phi \geq_\rho \Psi$, if $\Phi \vdash N : B$ and $(\Phi \vdash N) \in \mathcal{R}_B$ then $(\Phi \vdash [\rho]M'\ N) \in \mathcal{R}_C$ — by red. def.
Assume $\Phi \geq_\rho \Psi$, $\Phi \vdash N : B$ and $(\Phi \vdash N) \in \mathcal{R}_B$
$M \longrightarrow M'$ — by assumption
$M \longrightarrow^* V$ — by `s_trans`
$\Psi \vdash M$ halts — by def. of halts
$[\rho]M \longrightarrow [\rho]M'$ — by *subst. property*
$[\rho]M\ \ N \longrightarrow [\rho]M'\ \ N$ — by rule `s_app1`
$(\Phi \vdash [\rho]M'\ \ N) \in \mathcal{R}_C$ — by previous lines
$(\Phi \vdash [\rho]M\ \ N) \in \mathcal{R}_C$ — by i.h.
$(\Psi \vdash M) \in \mathcal{R}_{B \to C}$ — by definition of $\mathcal{R}$
□

Our aim is to show that all well-typed terms are reducible. As usual we must generalize this statement to well-typed open terms: If M is a well-typed term of type A under the typing assumptions $x_1{:}A_1, \ldots, x_n{:}A_n$ and σ is a substitution of the form $M_1/x_1,\ M_2/x_2,\ \ldots,\ M_n/x_n$ s.t. for all M_i we have $M_i{:}A_i$ in the context Φ, M_i is reducible (i.e. $(\Phi \vdash M_i) \in \mathcal{R}_{A_i}$) and $(\Phi \vdash [\sigma]M) \in \mathcal{R}_A$. Key to this generalization is the notion of a simultaneous substitution σ which provides well-typed reducible terms M_i for all the variables x_i of the term M. We call such a substitution a reducible substitution. It is worthwhile considering the domain and range of the reducible substitution carefully. What is the domain of such a reducible substitution? - It should be the typing context $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$. What should be the range of the reducible substitution? - It should be the typed neutral variable context $\Phi = y_1{:}B_1,\ y_1$ neut, $\ldots, y_k{:}B_k,\ y_k$ neut, as σ provides reducible terms M_i for each variable x_i and reducible terms are only meaningful in the typed neutral variable context. This highlights the subtle issues due to variables. Thinking of terms within a context of assumptions forces us to understand precisely what assumptions are necessary. We can now define reducible substitutions more precisely inductively on the structure of the typing context Γ.

$$\begin{array}{lcl} \mathcal{R}_{\cdot} & = & \{\Phi \vdash \cdot\ \} \\ \mathcal{R}_{\Gamma, x:A} & = & \{\Phi \vdash \sigma, M/x \mid \Phi \vdash M : A,\ (\Phi \vdash M) \in \mathcal{R}_A \text{ and } (\Phi \vdash \sigma) \in \mathcal{R}_\Gamma\ \} \end{array}$$

It is easy to see that if $(\Phi \vdash \sigma) \in \mathcal{R}_\Psi$ then the substitution σ is well-typed i.e. $\Phi \vdash \sigma : \Psi$. Before we prove the main lemma, we state two monotonicity properties about reducible terms and reducible substitutions.

Lemma 2.3 (Monotonicity Lemma).

1 If $(\Psi \vdash M) \in \mathcal{R}_A$ and $\Phi \geq_\rho \Psi$ then $(\Phi \vdash [\rho]M) \in \mathcal{R}_A$.
2 If $(\Psi \vdash \sigma) \in \mathcal{R}_{\Psi_0}$ and $\Phi \geq_\rho \Psi$ then $(\Phi \vdash [\rho]\sigma) \in \mathcal{R}_{\Psi_0}$.

Intuitively the monotonicity properties hold because the substitution ρ can only replace variables with neutral terms. This is guaranteed by the fact that ρ is a mapping between typed neutral variable contexts. As a consequence, proving that the term (or substitution) continues to be reducible is straightforward. We are now ready to state and prove the fundamental lemma.

Theorem 2.4 (Fundamental theorem). If $\Gamma \vdash M : A$ and $(\Psi \vdash \sigma) \in \mathcal{R}_\Gamma$ then $(\Psi \vdash [\sigma]M) \in \mathcal{R}_A$.

Proof. By induction on the typing derivation. We show only the interesting case:

Case $\dfrac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$

$\Psi \vdash [\sigma]N : A$ by *substitution lemma*
$\Psi \vdash [\sigma]M \in \mathcal{R}_{A\to B}$ and $\Psi \vdash [\sigma]N \in \mathcal{R}_A$ by i.h.
$\forall \Phi \geq_\rho \Psi, N'$ where $\Phi \vdash N' : A$
if $(\Phi \vdash N') \in \mathcal{R}_A$, then $(\Phi \vdash ([\rho][\sigma]M)\ \ N') \in \mathcal{R}_B$ by reducibility def.
$([\mathsf{id}_\Psi][\sigma]M)\ \ ([\sigma]N) \in \mathcal{R}_B$ by previous line choosing id_Ψ for ρ and Ψ for Φ
$([\mathsf{id}_\Psi][\sigma]M)\ \ ([\sigma]N) = ([\sigma]M)\ ([\sigma]N) = [\sigma](M\ N)$ by *subst. properties*
$[\sigma](M\ N) \in \mathcal{R}_B$ by previous lines

Case $\dfrac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \mathsf{lam}\ x.M : A \to B}$

$(\Psi \vdash \sigma) \in \mathcal{R}_\Gamma$ by assumption
$\Psi, x{:}A, x\ \mathsf{neut} \vdash \mathsf{id}_\Psi : \Psi$ by *weakening*
$(\Psi \vdash [\mathsf{id}_\Psi]\sigma) \in \mathcal{R}_\Gamma$ by monotonicity lemma
$\Psi, x{:}A, x\ \mathsf{neut} \vdash x : A$ by typing rule
$\Psi, x{:}A, x\ \mathsf{neut} \vdash x\ \mathsf{neut}$ by def. of neutral/normal terms
$(\Psi, x{:}A, x\ \mathsf{neut} \vdash x) \in \mathcal{R}_A$ by Main Lemma (2)
$(\Psi, x{:}A, x\ \mathsf{neut} \vdash [\mathsf{id}_\Psi]\sigma,\ x) \in \mathcal{R}_{\Gamma, x:A}$ by def. of reducibility of substitutions
$(\Psi, x{:}A, x\ \mathsf{neut} \vdash [[\mathsf{id}_\Psi]\sigma,\ x]M) \in \mathcal{R}_B$ by i.h.
$\Psi, x{:}A, x\ \mathsf{neut} \vdash [[\mathsf{id}_\Psi]\sigma,\ x]M$ halts by Main Lemma (1)
$\exists V.[[\mathsf{id}_\Psi]\sigma,\ x]M \longrightarrow^* V$ and $\Psi, x{:}A, x\ \mathsf{neut} \vdash V\ \mathsf{norm}$ by def. of halts
$\mathsf{lam}\ x.[[\mathsf{id}_\Psi]\sigma,\ x]M \longrightarrow^* \mathsf{lam}\ x.V$ by multi-step red. for lambda-abstractions
$[\sigma](\mathsf{lam}\ x.M) \longrightarrow^* \mathsf{lam}\ x.V$ by *subst. property*
$\Psi \vdash \mathsf{lam}\ x.V\ \mathsf{norm}$ by `n_lam`
$\Psi \vdash [\sigma](\mathsf{lam}\ x.M)$ halts by def. of halts
Given an extension $\Phi \geq_\rho \Psi$ and a term N, s.t. $\Phi \vdash N : A$ and $(\Phi \vdash N) \in \mathcal{R}_A$.
$(\Phi \vdash [\rho]\sigma) \in \mathcal{R}_\Gamma$ by monotonicity lemma
$(\Phi \vdash ([\rho]\sigma,\ N/x)) \in \mathcal{R}_{\Gamma, x:A}$ by def. of reducibility for substitutions
$(\Phi \vdash [\ [\rho]\sigma,\ N/x\ ]M) \in \mathcal{R}_B$ by i.h.
$(\Phi \vdash [N/x][[\rho]\sigma,\ x/x]M) \in \mathcal{R}_B$ by *subst. property*
$(\Phi \vdash (\mathsf{lam}\ x.[[\rho]\sigma,\ x/x]M)\ \ N) \in R_B$ by backwards closed lemma
$(\Phi \vdash [[\rho]\sigma](\mathsf{lam}\ x.M)\ \ N) \in R_B$ by *subst. properties*

$(\Phi \vdash [\rho]([\sigma](\mathsf{lam}\ x.M))\ \ N) \in R_B$ by *subst. properties*
$(\Psi \vdash [\sigma](\mathsf{lam}\ x.M)) \in R_{A\to B}$ by def. of reducibility
□

We made in the proof the use of weakening and strengthening explicit. Foreshadowing our mechanization, Beluga silently deals with the uses of weakening and strengthening. It also silently applies substitution properties to justify steps in the proof. As a consequence, the user can concentrate on the main aspects of the proof. We draw here in particular attention to the use of the substitution property to justify that $\mathsf{lam}\ x.[[\mathsf{id}_\Psi]\sigma,\ x]M$ is equivalent to $[\sigma](\mathsf{lam}\ x.M)$ which seems non-trivial at first.

Corollary 2.5 (Weak normalization). If $\Gamma \vdash M : A$ then M halts.

Proof. Let Γ be a context $x_1{:}A_1,\ \ldots,\ x_n{:}A_n$. Then there exists a substitution $\mathsf{id} = x_1/x_1,\ \ldots, x_n/x_n$ and a context $\Phi = x_1{:}A_1,\ x_1\ \mathsf{neut}, \ldots,\ x_n{:}A_n,\ x_n\ \mathsf{neut}$ s.t. $\Phi \vdash \mathsf{id} : \Gamma$. Moreover, by the main lemma, we have that $(\Phi \vdash [\mathsf{id}]M) \in \mathcal{R}_A$ and hence by reification $\Phi \vdash [\mathsf{id}]M$ halts. By *subst. property*, we have $[\mathsf{id}]M = M$ and therefore $\Phi \vdash M$ halts. □

In the proofs, we are drawing attention to the use of properties of simultaneous substitutions marking their use with italics. These properties are typically the source of the most overhead when formalizing results using various low level representations of variables and variable binding. In developing our definitions for well-typed terms, normal forms, and reducibility we have been guided by the contextual logical framework LF. Thinking about terms in the context they are meaningful in put into sharp focus the scope of terms and honed our mathematical thought processes providing a fresh perspective on the weak normalization proof using Kripke-style logical relations.

2.3. *Encoding of Lambda-Terms, Types, and Reductions in LF*

Next we demonstrate how the definitions of well-typed terms, normal forms, and reducibility translate directly and elegantly into Beluga. By hand we defined grammar and typing separately, but here it is actually more convenient to define intrinsically typed terms directly, as it obliterates the need to carry typing derivations separately and leads to a more compact mechanization. Below, `tm` defines our family of simply-typed lambda terms indexed by their type as an LF signature. In typical higher-order abstract syntax fashion, lambda abstraction takes a *function* representing the abstraction of a term over a variable. There is no case for variables, as they are treated implicitly. We remind the reader that this is a weak, representational function space – there is no case analysis or recursion, and hence only genuine lambda terms can be represented.

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

```
LF tm : tp → type =
| app : tm (arr T S) → tm T → tm S
| lam : (tm T → tm S) → tm (arr T S);
```

We then encode our step relation in a similar type-preserving fashion. Note in particular the use of LF application to encode the object-level substitution in the `s_beta` case.

```
LF step  : tm A → tm A → type =
| s_beta : step (app (lam λx.M x) N) (M N)
```

```
| s_lam  : ({x:tm A} step (M x) (M' x)) → step (lam λx.M x) (lam λx.M' x)
| s_app1 : step M M' → step (app M N) (app M' N)
| s_app2 : step N N' → step (app M N) (app M N')
;

LF mstep : tm A → tm A → type =
| s_refl : mstep M M
| s_trans: step M M' → mstep M' M'' → mstep M M''
;
```

Finally, we encode the judgments M norm and M neut as type families `normal` and `neutral` in LF in the standard fashion. The most interesting case is the encoding for `normal (lam λx.M x)`. A term `lam λx.M x` is normal, if we can show that for all `x:tm` and `neutral x`, `normal (M x)`. It directly translates the parametric and hypothetical rule using the LF function space.

```
LF normal : tm A → type =
| n_lam : ({x:tm A} neutral x → normal (M x)) → normal (lam λx.M x)
| n_neut : neutral M → normal M
and neutral : tm A → type =
| n_app : neutral M → normal N → neutral (app M N);
```

2.4. *Encoding Logical Relations as Inductive Definitions*

The reducibility predicate cannot be directly encoded as a type family in LF, since it involves a strong, computational function space. Moreover, our earlier definition of reducibility for open terms relied on reasoning about context extensions. In the logical framework LF, the context of assumptions is ambient and implicit. This makes encoding of logical relations in LF challenging. Luckily, Beluga does not only supports LF specifications, but provides a first-order logic with inductive definitions [CP12] to express properties about contexts, contextual objects, and substitutions. This is key to stating properties about well-typed terms that depend on typing assumptions. This is accomplished by pairing the well-typed term together with its context using contextual types [NPP08]. For example, lam $y.x\ y$ where x has type $\mathbf{i} \to \mathbf{i}$ is represented as the contextual LF term `[x:tm (arr i i) ⊢ lam λy.app x y]`. If a term is closed, for example `[ ⊢ lam λx. x]`, we simply write `[lam λx.x]` omitting the turnstyle to improve readability.

When stating reducibility candidates we rely on the typed neutral variable context which in addition to typing assumptions for variables also carries an assumption stating that variables are neutral. As we have seen, it is often convenient to abstract over the concrete context of assumptions to express meaningful properties. In Beluga, we can classify contexts using context schemas which resemble world declarations in Twelf. Schema declarations in Beluga express an invariant we want to hold when stating inductive properties about LF objects. The schema declaration `ctx` states that a context of schema `ctx` contains assumptions that are instances of `tm t`. For example, the context `x:tm i, y:tm (arr i i)` is a valid typing context of schema `ctx`. On the other hand the context `a:tp, x:tm a` does not fit the schema `ctx`. The schema declaration `nctx` states that a context must consist of a block of two assumptions, namely `x:tm t` and `n_x: neut x`. It encodes our definition of a neutral variable context from page 5 forcing the fact that the

typing assumption `x:tm t` is introduced together with the assumption `neut x` which states that the variable `x` is neutral. Formally, a block is a Σ-type.

```
schema ctx  = some [t:tp] tm t;
schema nctx = some [t:tp] block x:tm t, n_x:neut x;
```

In our previous reducibility definition, we stated that a term `M` in a typed neutral variable context Ψ is reducible at type $A \to B$ if for all extensions $\Phi \geq_\rho \Psi$ where ρ is a neutral substitution and well-typed terms N where $(\Phi \vdash N) \in \mathcal{R}_A$, we have $(\Phi \vdash [\rho]M\ N) \in \mathcal{R}_B$. We now can translate directly the reducibility definition given earlier on page 6 into an indexed recursive type.

In contrast to LF type families which have kind **type**, we define recursive types using either keyword **inductive** or the keyword **stratified** and declare its kind using the kind **ctype**. We first state the inductive predicate `Halts` which is indexed by the context ψ of schema `nctx`, the closed type `A` and a term `M` of type `A` in the context ψ.

```
inductive Halts : (ψ:nctx) {A:[tp]}{M:[ψ ⊢ tm A[]]} ctype =
| Halts : {V:[ψ ⊢ tm _]} [ψ ⊢ mstep M V] → [ψ ⊢ normal V] → Halts [A] [ψ ⊢ M];

stratified Reduce : (ψ:nctx) {A:[tp]}{M:[ψ ⊢ tm A[]]} ctype =
| Base : Halts [ ⊢ i] [ψ ⊢ M] → Reduce [i] [ψ ⊢ M]
| Arr :  {M:[ψ ⊢ tm (arr A[] B[])]}
         Halts [ ⊢ arr A B] [ψ ⊢ M] →
         ({φ:nctx} {ρ:[φ ⊢ ψ]} {N:[φ ⊢ tm A[]]}
            Reduce [A] [φ ⊢ N] → Reduce [ ⊢ B ] [φ ⊢ app M[ρ] N])
         → Reduce [arr A B] [ψ ⊢  M];
```

By wrapping the context declaration in round parenthesis (in contrast to curly parenthesis) when we declare the kind of an inductive type, we express that ψ remains implicit in the use of this type family and may be omitted, where curly brackets would denote an explicit dependent argument. In other words, `(ψ:nctx)` is simply a type annotation stating the schema the context ψ must have, but we do not need to pass an instantiation for ψ explicitly. We refer to variables occurring inside a contextual object (i.e. inside `[ ]`) as meta-variables to distinguish them from bound variables. In general, all meta-variables such as `A`, `M`, `V`, `N`, etc. are associated with a postponed substitution which may be omitted, if it is the identity substitution. As `A` is closed, we must weaken it by applying the empty substitution `[]` when we state the type of `M` as $\psi \vdash$ `tm A[]`.

We can then read the inductive definition of `Halts` as follows: `Halts [⊢ A] [ψ ⊢ M]` if the term `M` steps to a normal term `V` (i.e. `[ψ ⊢ mstep M V]`) of the same type `A`.

Second, we encode the reducibility predicate using the stratified type `Reduce`. Stratified types are defined inductively over one of its indices - in our case we define `Reduce` inductively on the type `A`. Stratified types do not give directly rise to an induction principle – instead, we typically reason on the inductive argument directly.

For base types, we simply state that `[ψ ⊢ M]` is reducible at base type `i`, if `[ψ ⊢ M]` halts. For function types, we state that `[ψ ⊢ M]` is reducible at type `arr A B`, if it halts and for all contexts ϕ, substitutions ρ from ψ to ϕ, and terms `N` of type `A` in the context ϕ that are reducible, we have that $\phi \vdash$ `app M[ρ] N` is reducible at type `B`.

As both ψ and ϕ must satisfy the context schema `nctx`, we are guaranteed that ρ is a substitution providing neutral terms for all the variables in ψ.

In declaring LF types and type families as well as inductive type families, we left some variables free. As in Twelf [PS99], Beluga's type reconstruction infers types for any free variable in a given type or kind declaration and implicitly quantifies over them [Pie13,FP14]. Programmers subsequently do not need to supply arguments for implicitly quantified variables.

2.5. *Encoding Inductive Proofs as Total Functions in Beluga*

Inductive proofs can be encoded as total functions using pattern matching in Beluga. We begin with encoding the lemma showing that halts is closed under expansion (Lemma 2.2). The lemma was proven by induction on the type of the term and hence our corresponding function `closed` is written by case-analysis and pattern matching on the type. We specify informally the theorem as a type in Beluga which can then be easily translated into the actual type of function `closed`:

For all closed types `A`, terms `M` and `M'` of type `A` in the context ψ,
for all reductions `S:`$[\psi \vdash$ `step M M']`, if `Reduce [⊢A]` $[\psi \vdash$ `M']` then `Reduce [⊢A]` $[\psi \vdash$ `M]`

Recall that we explicitly quantify over objects using curly braces. Beluga's reconstruction algorithm can often infer the type of free variables, and abstracts over these variables at the outside. As however, terms depend on types we cannot leave Beluga's reconstruction algorithm to infer their types and we must specify them explicitly.

```
rec closed : {A:[⊢ tp]}{M:[ψ ⊢ tm A[]]}{M':[ψ ⊢ tm A[]]}
             {S:[ψ ⊢ step M M']} Reduce [A] [ψ ⊢ M'] → Reduce [A] [ψ ⊢ M] =
/ total a (closed _ a) /
Λ A,M,M',S ⇒ fn r ⇒ case [A] of
| [⊢ i] ⇒
   let Base (Halts [ψ ⊢ V] [ψ ⊢ S'] v) = r in
     Base (Halts [ψ ⊢ V] [ψ ⊢ s_trans S S'] v)
| [⊢ arr _ C] ⇒
  let Arr [ψ ⊢ M'] (Halts [ψ ⊢ V] [ψ ⊢ S'] v) f = r in
  Arr [ψ ⊢ M] (Halts [ψ ⊢ V] [ψ ⊢ s_trans S S'] v)
  (Λ φ, ρ, N ⇒ fn rn ⇒ closed [C] [φ ⊢ app M[ρ] N] [φ ⊢ app M'[ρ] N]
                              [φ ⊢ s_app1 S[ρ]] (f [φ] [φ ⊢ ρ] [φ ⊢ N] rn))
;
```

The proof itself then proceeds by pattern matching on the type `A`. In the base case where `A=i`, we model inversion in the actual proof using pattern matching on the definition of `Reduce` and `Halt` obtaining a derivation `S'` for $[\psi \vdash$ `step M' V]` and a proof `v` that `V` is normal. We then use the assumption `S` which stands for $[\psi \vdash$ `step M M']` and `S'`, and build a derivation for $[\psi \vdash$ `step M V]` using transitivity (i.e. the rule `s_trans`). From this, it is now trivial to provide a witness that `Reduce [⊢ i]` $[\psi \vdash$ `M]`.

If `A` = `arr B C`, we pattern match on `Reduce [ ⊢ arr B C]` $[\psi \vdash$`M']` obtaining a function `f` which says "for all ϕ, ρ, N, if `Reduce [⊢ B]` $[\phi \vdash$ `N]` then `Reduce [⊢ C]` $[\phi \vdash$ `app M'[`ρ`] N]`. This models the inversion step in the proof.

We now build a witness for `Reduce [⊢ C]` $[\phi \vdash$ `app M[`ρ`] N]` by building a function that assumes ϕ, ρ, and well-typed term `N` as well as `rn` which stands for `Reduce [⊢ B]` $[\phi \vdash$ `N]` and returns `Reduce [⊢ C]` $[\phi \vdash$ `app M[`ρ`] N]` following our earlier proof arguing by induction

(recursion). Beluga checks that the function is total and hence represents a proof. In specifying the proof we explicitly passed instantiations when making our recursive call. We could have left these with an underscore leaving reconstruction to figure out the appropriate instantiation. However, we feel the proof becomes clearer when specifying them explicitly.

The encoding of the the main lemma follows directly the on-paper proof. The statement of the lemmas is encoded as a type in Beluga making explicit the quantification over the context ψ, the type `A`, and the term `R`. We again proceed by induction on the type `A`. We only show the representation of the statement and omit the implementation of the proof here, but it can be found in the electronic appendix. The appeal to weakening and strengthen which we highlighted in the on paper proof are unnecessary in the mechanization as Beluga silently takes this into consideration.

```
rec main1 :{ψ:nctx}{A:[tp]}{M:[ψ ⊢ tm A[]]} Reduce [A] [ψ ⊢ M] → Halts [A] [ψ ⊢ M]
and main2:{ψ:nctx}{A:[tp]}{R:[ψ ⊢ tm A[]]}{NR:[ψ ⊢ neut R]} Reduce [A] [ψ ⊢ R]
```

We now must state precisely what it means for a substitution to be reducible. We do this by employing another indexed recursive type: a predicate expressing that the substitution was built up as a list of reducible terms. The notation σ stands for a substitution variable. Its type is written `[`$\phi \vdash \gamma$`]`, meaning that it has domain γ and range ϕ, i.e. it takes variables in γ to terms of the same type in the context ϕ. In the base case, the empty substitution is reducible. In the `Cons` case, we read this as saying: if σ is a reducible substitution (implicitly at type `[`$\phi \vdash \gamma$`]`) and `M` is a reducible term at type `A`, then σ with `M` appended is a reducible substitution (implicitly at type `[`$\phi \vdash \gamma$`,x:tm A[]]` – the domain has been extended with a variable of type `A`).

```
datatype LogSub : {γ:ctx}(φ:nctx){σ:[φ ⊢ γ]} ctype =
| Nil : LogSub [] [φ ⊢ ^ ]
| Dot : LogSub [γ] [φ ⊢ σ] → Reduce [A] [φ ⊢ M]
     → LogSub [φ ⊢ σ, M ];
```

We explicitly quantify over γ, the domain of the substitution, and leave the range of the substitution implicit by using round parenthesis writing `(`ϕ`:nctx)`; we explicitly quantify over substitution variables using curly braces writing `{`σ`:[`$\phi \vdash \gamma$`]}`. There are two monotonicity lemmas we rely on in our proof of the fundamental lemma. These lemmas correspond exactly to those we used in the on-paper proof.

1 Given a neutral substitution σ with domain ψ and range ϕ, if `Reduce [⊢ A] [`$\psi \vdash$ `M]` then `Reduce [A] [`$\phi \vdash$ `M[`σ`] ]`

```
reduce_monotone: (φ:nctx){σ:[φ ⊢ ψ]} Reduce [A] [ψ ⊢ M] → Reduce [A] [φ ⊢ M[σ]]
```

2 Given a neutral substitution ρ with domain ψ and range ϕ and a reducible substitution σ from γ to ψ, we know that the composition $\sigma[\rho]$ is a reducible substitution from γ to ϕ.

```
redsub_monotone: {ρ:[φ ⊢ ψ]} LogSub [γ] [ψ ⊢ σ] → LogSub [γ] [φ ⊢ σ[ρ]]
```

Finally, our main lemma is standard and takes the form we would expect from the handwritten proof: if `M` is a well-typed term in the typing context γ, and σ is a reducible substitution with domain γ and range ψ which provides neutral terms as instantiations for each of the free variables of `M`, then `M[`σ`]` (that is, the application of σ to `M`) is reducible. We proceed by induction on the term. When it is a variable, we appeal to `redvar_monotone` which is a special case of monotonicity lemma `reduce_monotone` where `M` is a variable. When it is an application, we straightforwardly apply the functional argument we obtain from the induction hypothesis for `M1` to the induction hypothesis for `M2`. The application case is straightforward thanks to the equational theory of substitutions supported in Beluga. The `lam` case is the most interesting, however it follows directly the on paper proof. Revisiting the on-paper proof, we note that we did not have to concern ourselves with the property of substitutions that we wrote explicitly in the paper proofs. In Beluga, normalizing LF objects incorporates the equational theory of simultaneous substitutions and applies the necessary substitution properties automatically.

```
rec fund : {M:[γ ⊢ tm A[]]} LogSub [γ] [ψ ⊢ σ] → Reduce [A] [ψ ⊢ M[σ]] =
/ total m (fund γ ψ a σ m) /
Λ M ⇒ fn rs ⇒ let (rs : LogSub [γ] [ψ ⊢ σ] ) = rs in
 case [γ ⊢ M] of
| [γ ⊢ #p] ⇒ redvar_monotone [γ] [γ ⊢ #p] rs
| [γ ⊢ app M1 M2]  ⇒
  let Arr [ψ ⊢ _ ] h f = fund [γ ⊢ M1]  rs in
   f [ψ] [ψ ⊢ …] [ψ ⊢ M2[σ]] (fund [γ ⊢ M2] rs)
| [γ ⊢ lam λx. M1] ⇒
   let rx = reflect [ϕ,b:block x:tm _,y:neutral x] [⊢ _] [ϕ,b ⊢ b.1] [ϕ,b ⊢ b.2] in
   let q0 = eval [γ,x:tm _ ⊢ M1]
                 (Dot (monotoneSub [ϕ,b:block x:tm _,y:neutral x ⊢ …] rs) rx) in
   let Halts [ϕ,b:block x:tm A1[],y:neutral x ⊢ _]
             [ϕ,b:block x:tm A1[],y:neutral x ⊢ MS]
             [ϕ,b:block x:tm A1[],y:neutral x ⊢ NV]
     = reify [ϕ,b:block x:tm _,y:neutral x] [⊢ _] [ϕ,b ⊢ M1[σ[…],b.1]] q0 in
   Arr [ϕ ⊢ lam (λx. M1[σ[…], x])]
   (Halts [_ ⊢ _]
     (m_lam [ϕ, x:tm A1[]  ⊢ M1[σ[…],x]]  [ϕ, b:block (x:tm A1[], y:neutral x) ⊢ MS])
     [ϕ ⊢ n_lam (λx.λy. NV[…,<x;y>])])
   (Λ ψ,ρ,N ⇒ fn rN ⇒
     closed [⊢ _ ] [ψ ⊢ app (lam λx.M1[σ[ρ[…]],x]) N ] [ψ ⊢ M1[σ[ρ[…]], N] ]
            [ψ ⊢  s_beta] (eval [γ,x:tm _ ⊢  M1] (Dot (monotoneSub [ψ ⊢ ρ] rs) rN)))
```

Weak normalization is now a trivial corollary. We only need to show that given a term the context γ there always exist a neutral identity substitution.

Our development is ≈180 lines, and it follows the handwritten proof very closely, with essentially no extra overhead. Compared to low-level techniques for variable binding, it is a huge win to not be burdened with proving properties of simultaneous substitution. The approach scales naturally when we also consider not only β-reduction, but also η-expansion. Even in other systems employing higher-order abstract syntax, such as Abella [Gac08], simultaneous substitution is not a first-class notion, and must be defined explicitly. This means that one must still prove properties of simultaneous substitution, although it is somewhat easier in Abella than with a low-level representation because simultaneous substitution can be built out of the provided individual substitu-

$\boxed{M \equiv N : A}$ Terms M and N are declaratively equivalent at type A

$$\frac{M_1 \equiv N_1 : B \to B \quad M_2 \equiv N_2 : B}{M_1\ M_2 \equiv N_1\ N_2 : A}\ \texttt{d\_app} \qquad \frac{\begin{array}{c}\overline{x : A}^{\ u} \\ \vdots \\ M\ x \equiv N\ x : B\end{array}}{\mathsf{lam}\ x.M \equiv \mathsf{lam}\ x.N : A \to B}\ \texttt{d\_lam}^{x,u}$$

$$\frac{\begin{array}{c}\overline{x : A}^{\ u} \\ \vdots \\ M_1\ x \equiv N_1\ x : B\end{array} \quad M_2 \equiv N_2 : A}{(\mathsf{lam}\ x.M_1)\ M_2 \equiv [N_2/x]N_1 : B}\ \texttt{d\_beta}^{x,u} \qquad \frac{\begin{array}{c}\overline{x : A}^{\ u} \\ \vdots \\ M\ x \equiv N\ x : B\end{array}}{M \equiv N : A \to B}\ \texttt{d\_ext}^{x,u}$$

$$\frac{M : A}{M \equiv M : A}\ \texttt{d\_refl} \qquad \frac{N \equiv M : A}{M \equiv N : A}\ \texttt{d\_sym} \qquad \frac{M \equiv L : A \quad M \equiv N : A}{M \equiv N : A}\ \texttt{d\_trans}$$

Fig. 1. Declarative Equivalence

tion. However, one must still explicitly prove that the defined simultaneous substitution is a congruence in Abella, i.e. $([\sigma]M)\ ([\sigma]N) = [\sigma](M\ N)$ and similarly for the more complex λ-abstraction case. In our system, this burden is lifted completely.

For systems such as Twelf [PS99] and Delphin [PS08] a direct formulation of normalization proofs is out of reach, since they lack first-class contexts and recursive types. Instead [SS08] proposed to represent and reason about an auxiliary logic to overcome the limited meta-logical strength of systems such as Twelf. However, to our knowledge this technique has not been employed to prove properties about open terms which are significantly more complicated.

More generally, our approach should prove useful for many (Kripke) logical relations proofs, such as parametricity, full abstraction, or various kinds of completeness proofs. This is especially so for larger languages (e.g. with case expressions) where one must use such properties of substitution repeatedly.

3. Example: Completeness of Algorithmic Equality

In this section we consider the completeness proof of algorithmic equality for simply-typed lambda-terms. 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.

3.1. *The Basics: Declarative and Algorithmic Equality*

We revisit again the simply typed lambda-calculus from the previous section. However here we only only perform weak head reduction. This makes algorithmic equality more efficient, and also simplifies many aspects of the proof. We concentrate here on giving the

motivation and high level structure of the completeness proof for algorithmic equality. For more detail, we refer the reader to [Cra05] and [HP05].

Declarative equivalence (see Fig. 1) includes convenient but non-syntax directed rules such as transitivity and symmetry, among rules for congruence, extensionality and β-contraction. In particular, it declares a term M equal to itself at type A, provided it is well-typed. It may also include type-directed rules such as extensionality at unit type:

$$\frac{M : \text{Unit} \quad N : \text{Unit}}{M \equiv N : \text{Unit}}$$

This rule relies crucially on type information, so the common untyped rewriting strategy for deciding equivalence no longer applies. Instead, one can define an algorithmic notion of equivalence which is directed by the syntax of types. This is the path we follow here. 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 $x\, M_1 \ldots M_n$ (see Fig. 3.1).

$\boxed{M \Leftrightarrow N : A}$ Terms M and N are algorithmically equivalent at type A

$\boxed{M \leftrightarrow N : A}$ Paths M and N are algorithmically equivalent at type A

$$\frac{M \longrightarrow^* M' \quad N \longrightarrow^* N' \quad M' \leftrightarrow N' : \mathbf{i}}{M \Leftrightarrow N : \mathbf{i}}\ \texttt{alg\_base} \qquad \frac{\begin{array}{c}\overline{x \leftrightarrow x : A}\ a_x \\ \vdots \\ M\ x \Leftrightarrow N\ x : B\end{array}}{M \Leftrightarrow N : A \to B}\ \texttt{alg\_arr}^{x,a_x}$$

$$\frac{M_1 \leftrightarrow M_2 : A \to B \quad N_1 \Leftrightarrow N_2 : A}{M_1\ N_1 \leftrightarrow M_2\ N_2 : B}\ \texttt{alg\_app}$$

Fig. 2. Algorithmic Equivalence

3.2. *Hand-written Proof Outline: Completeness of Algorithmic Equality*

In what follows, we sketch the proof of completeness of algorithmic equivalence for declarative equivalence. A direct proof by induction over derivations fails unfortunately in the application case where we need to show that applying equivalent terms to equivalent arguments yields equivalent results. Instead, one can proceed by proving a more general statement that declaratively equivalent terms are *logically equivalent*, and so in turn algorithmically equivalent. Logical equivalence is a relation defined directly on the structure of the types. It relates well-typed terms, as declarative equality is in fact only defined on well-typed terms. In the base case two terms M and N are logically related at base type, if they are algorithmically equal. As algorithmic equality relies on assumptions $x \leftrightarrow x : A$, we define logically equivalent terms in a well-typed neutral algorithmic equality context where we keep the typing assumptions, i.e. $x : A$, and the fact that every variable is algorithmically equal to itself, i.e. $x \leftrightarrow x : A$.

$$\text{Well-Typed Algorithmic Equality Context } \Psi \quad ::= \quad \cdot \mid \Psi,\ x : A,\ n_x : x \leftrightarrow x : A$$

It might seem redundant to keep typing assumptions, however the completeness proof of algorithmic equality relies on this information in subtle ways. We therefore define

$$(\Psi \vdash M \approx N) \in \mathcal{R}_A \quad \text{Terms } M \text{ and } N \text{ are logically equivalent at type } A$$

The key case is at function type, which directly defines logically equivalent terms at function type as taking logically equivalent arguments to logically equivalent results. Algorithmic equality for terms M and N of type $A \to B$ states that it suffices to compare their application to *fresh* variables: Assuming that $x \leftrightarrow x : A$, we show that $M\ x \Leftrightarrow N\ x : B$. We hence must be able to reason about future extensions of the initial context. This Kripke-style monotonicity condition hence naturally arises and is one of the reasons that this proof is challenging. While in the previous weak normalization proof for simply typed lambda-terms this quantification over context extensions can often be avoided using other technical tricks, it is hard to avoid in the completeness proof of algorithmic equality.

Following the footprint of our previous reducibility definition, we use the simultaneous substitution π as a witness to move between two algorithmic equality contexts. This again ensures that we only replace variables with neutral terms thereby guaranteeing monotonicity. In the course of the completeness proof, π will actually only ever be instantiated by substitutions which simply perform weakening, i.e. replacing variables by variables. We call such a substitution π a *path substitution*. Our definition of logically equivalent terms generalizes Crary's definition by witnessing the context extension by path substitutions.

$$\begin{array}{lcl}
\mathcal{R}_{\mathbf{i}} & = & \{\Psi \vdash M_1 \approx M_2 \mid \Psi \vdash M_1 \Leftrightarrow M_2 : \mathbf{i}\} \\
\mathcal{R}_{A \to B} & = & \{\Psi \vdash M_1 \approx M_2 \mid \forall \Phi \geq_\pi \Psi,\ N_1,\ N_2 \text{ such that } \Phi \vdash N_1 : A \text{ and } \Phi \vdash N_2 : A, \\
 & & \quad \textit{if } (\Phi \vdash N_1 \approx N_2) \in \mathcal{R}_A \textit{ then } (\Phi \vdash [\pi]M_1\ N_1 \approx [\pi]M_2\ N_2) \in \mathcal{R}_B\}
\end{array}$$

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 3.1 (Monotonicity).
If $(\Psi \vdash M \approx N) \in \mathcal{R}_A$ and $\Phi \vdash \pi : \Psi$, then $(\Phi \vdash [\pi]M \approx [\pi]N) \in \mathcal{R}_A$

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

Lemma 3.2 (Weak Head Closure under Expansion).
If $(\Psi \vdash N_1 \approx N_2) \in \mathcal{R}_A$ and $M_1 \longrightarrow^* N_1$ and $M_2 \longrightarrow^* N_2$ then $(\Psi \vdash M_1 \approx M_2) \in \mathcal{R}_A$

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 3.3 (Main lemma).

1 If $\Psi \vdash M \leftrightarrow N : A$ then $(\Psi \vdash M \approx N) \in \mathcal{R}_A$
2 If $(\Psi \vdash M \approx N) \in \mathcal{R}_A$ then $\Psi \vdash M \Leftrightarrow N : A$

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 for path equivalence. We will not go into detail about these lemmas, as they are relatively mundane, but refer the reader to the discussion in [Cra05].

What remains is to show that declarative equivalence implies logical equivalence. This requires a generalization of the statement to all instantiations of open terms by related substitutions. If σ_1 is of the form $M_1/x_1, ..., M_n/x_n$ and σ_2 is of the form $N_1/x_1, ..., N_n/x_n$ and Γ is of the form $x_1{:}A_1, ..., x_n{:}A_n$, and each M_i (and N_i resp.) has type A_i, we write $\Delta \vdash \sigma_1 \approx \sigma_2 : \Gamma$ to mean that $\Delta \vdash M_i \approx N_i : A_i$ for all i. Note that $\sigma_1 \approx \sigma_2$ relates the typing context Γ to the well-typed neutral algorithmic equality context Δ.

$$\begin{aligned}\mathcal{R}_{\cdot} &= \{\Phi \vdash \cdot \approx \cdot\ \} \\ \mathcal{R}_{\Gamma,x:A} &= \{\Phi \vdash (\sigma_1, M_1/x) \approx (\sigma_2, M_2/x) \mid \Phi \vdash M_1 : A,\ \Phi \vdash M_2 : A,\ \text{and} \\ &\qquad\qquad (\Phi \vdash \sigma_1 \approx \sigma_2) \in \mathcal{R}_\Gamma\ \}\end{aligned}$$

Theorem 3.4 (Fundamental theorem).
If $\Gamma \vdash M \equiv N : A$ and $(\Delta \vdash \sigma_1 \approx \sigma_2) \in \mathcal{R}_\Gamma$ then $(\Delta \vdash M[\sigma_1] \approx N[\sigma_2]) \in \mathcal{R}_A$

Proof. The proof goes by induction on the derivation of $\Gamma \vdash M \equiv N : A$. We show one interesting case in order to demonstrate some sources of complexity.

Case: $\dfrac{\Gamma, x : A \vdash M_1 \equiv M_2 : B}{\Gamma \vdash \lambda x.M_1 \equiv \lambda x.M_2 : A \Rightarrow B}$

$(\Delta \vdash \sigma_1 \approx \sigma_2) \in \mathcal{R}_\Gamma$ by assumption
Given an extension $\Delta' \geq_\pi \Delta$ and terms N_1, N_2 s.t.
$\Delta' \vdash N_1 : A$ and $\Delta' \vdash N_2 : A$ and $(\Delta' \vdash N_1 \approx N_2) \in \mathcal{R}_A$
$(\Delta' \vdash \sigma_1[\pi] \approx \sigma_2[\pi]) \in \mathcal{R}_\Gamma$ by monotonicity
$(\Delta' \vdash (\sigma_1[\pi], N_1/x) \approx (\sigma_2[\pi], N_2/x)) \in \mathcal{R}_{\Gamma,x:A}$ by definition
$(\Delta' \vdash M_1[\sigma_1[\pi], N_1/x] \approx M_2[\sigma_2[\pi], N_2/x]) \in \mathcal{R}_B$ by induction hypothesis
$(\Delta' \vdash M_1[\sigma_1[\pi], x/x][N_1/x] \approx M_2[\sigma_2[\pi], x/x][N_2/x]) \in \mathcal{R}_B$ by *substitution properties*
$(\Delta' \vdash (\lambda x.M_1[\sigma_1[\pi], x/x])\ N_1 \approx (\lambda x.M_2[\sigma_2[\pi], x/x])\ N_2) \in \mathcal{R}_B$ by weak head closure
$(\Delta' \vdash ((\lambda x.M_1)[\sigma_1])[\pi]\ N_1 \approx ((\lambda x.M_2)[\sigma_2])[\pi]\ N_2) \in \mathcal{R}_B$ by *substitution properties*
$(\Delta \vdash (\lambda x.M_1)[\sigma_1] \approx (\lambda x.M_2)[\sigma_2]) \in \mathcal{R}_{A\to B}$ by definition of logical equivalence
□

We observe that this proof relies heavily on equational properties of substitutions. Some of this complexity appears to be due to our choice of quantifying over substitutions

$\Delta \vdash \pi : \Gamma$ instead of extensions $\Delta \geq \Gamma$. However, we would argue that reasoning instead about extensions $\Delta \geq \Gamma$ does not remove this complexity, but only rephrases it.

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

Corollary 3.5 (Completeness). If $\Gamma \vdash M \equiv N : A$ then $\Gamma \vdash M \Leftrightarrow N : A$

3.3. *Encoding Lambda-terms, Typing and Reduction in the Logical Framework LF*

Unlike the previous case study where we defined intrinsically terms, we define here untyped lambda-terms and weak-head reduction on untyped terms in LF employing HOAS for the representation of lambda abstraction and defining typing rules for terms using an LF type family `oft`.

```
LF tm : type =                  LF oft : tm → tp → type =
| app : tm → tm → tm           | t_app : oft M (arr A B) → oft N A → oft (app M N) B
| lam : (tm → tm) → tm;        | t_lam : ({x:tm} oft x A → oft (M x) B)
                                        → oft (lam λx. M x) (arr A B);
```

Weak head reduction adopts a call-by-name strategy (i.e. we drop the rule `s_app2`) and multi-step reductions remains unchanged.

3.4. *Encoding Declarative and Algorithmic Equivalence*

We now encode declarative and algorithmic equivalence of terms in LF using higher-order abstract syntax. Parametric and hypothetical derivations are again mapped to LF function spaces, but are straightforward.

```
LF deq : tm → tm → tp → type =
| d_beta : ({x:tm} oft x T → deq (M2 x) (N2 x) S) → deq M1 N1 T
           → deq (app (lam λx. M2 x) M1) (N2 N1) S
| d_lam  : ({x:tm} oft x T → deq (M x) (N x) S)
          → deq (lam λx. M x) (lam λx. N x) (arr T S)
| d_ext  : ({x:tm} oft x T → deq (app M x) (app N x) S)
         → deq M N (arr T S)
| d_app  : deq M1 M2 (arr T S) → deq N1 N2 T → deq (app M1 N1) (app M2 N2) S
| d_refl : oft M T → deq M M T
| d_sym  : deq M N T → deq N M T
| d_trans: deq M N T → deq N O T → deq M O T;
```

Algorithmic equality of terms is defined as two mutually recursive LF specifications. We write `algeq M N T` for algorithmic equivalence of normal terms `M` and `N` at type `T` and `algeqNeu P Q T` for algorithmic path equivalence at type `T` – these are terms whose head is a variable, not a lambda abstraction. Following the previous on-paper definition 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 ⇒ S` if applying them to a fresh variable `x` of type `T` yields equivalent terms. 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.

```
LF algeq: tm → tm → tp → type =
| alg_base: mstep M P → mstep N Q → algeqNeu P Q i
         → algeq M N i.
| alg_arr : ({x:tm} algeqNeu x x T → algeq (app M x) (app N x) S)
        → algeq M N (arr T S)

and algeqNeu : tm → tm → tp → type
| alg_app : algeqNeu M1 M2 (arr T S) → algeq N1 N2 T
         → algeqNeu (app M1 N1) (app M2 N2) S;
```

By describing algorithmic and declarative equality in LF, we gain structural properties and substitution for free. For this particular proof, only weakening is important.

Two different forms of contexts are relevant for this proof. We describe these with **schema** definitions in Beluga. Below, we define the schema `actx`, which enforces that term variables come paired with a typing assumption `oft x T` and an algorithmic equality assumption `algeqNeu x x t` for some type `t`. In addition, we define the schema `ctx` of typing contexts.

```
schema actx = some [t:tp] block x:tm, t_x: oft x t, a_x:algeqNeu x x t;
schema ctx  = some [t:tp] block x:tm, t_x: oft x t;
```

3.5. *Encoding Logical Equivalence as Inductive Definition*

To define logical equivalence, we need the notion of path substitution mentioned in Sec. 3.2. For this purpose, we use again Beluga's built-in notion of simultaneous substitutions. We write `[`$\phi \vdash \psi$`]` for the built-in type of simultaneous substitutions which provide for each variable in the context ψ a corresponding term in the context ϕ. When ψ is of schema `actx` , such a substitution consists of blocks of the form `<M;D;P>` where `M` is a term, `D` stands for a typing derivation that M is well-typed, and `P` is a proof showing that `M` is algorithmically equal to itself and is in fact a neutral term.

To achieve nice notation, we define an LF type of pairs of terms, where the infix operator $\approx$ simply constructs a pair of terms:

```
LF tmpair : type =
| ≈ : tm → tm → tmpair % infix;
```

Logical equivalence, written `Log [`$\psi \vdash$ `M ≈ N] [A]`, expresses that `M` and `N` are logically related in context ψ at type A and is again defined as a stratified type. Beluga verifies that this stratification condition is satisfied. In this case, the definition is structurally recursive on the type `A`.

```
stratified Log : (ψ:actx) [ψ ⊢ tmpair] → [tp] → ctype =
| LogBase :  [ψ ⊢ algeq M N i] → Log [ψ ⊢ M ≈ N]  i]
| LogArr  : {M1:[ψ ⊢ tm]}{M2:[ψ ⊢ tm]}
             ({φ:actx}{π:[φ ⊢ ψ]}
             {N1:[φ ⊢ tm]}{D1:[φ ⊢ oft N1 T[]]}{N2:[φ ⊢ tm]}{D2:[φ ⊢ oft N2 T[]]}
                Log [φ ⊢ N1 ≈ N2] [T] → Log [φ ⊢ app M1[π] N1 ≈ app M2[π] N2] [S])
             → Log [ψ ⊢ M1 ≈ M2] [arr T S];
```

At base type, two terms are logically equivalent if they are algorithmically equivalent. At arrow type we employ the monotonicity condition mentioned in Sec. 3: `M1` is related to `M2` in Ψ if, for any extension $\Phi \geq_\pi \Psi$, and terms `N1`, `N2` that are well-typed in Φ, and logically related in Φ, we have that `app M1[`π`] N1` is related to `app M2[`π`] N2` in Φ. In the kind of `Log`, we quantify `(`ψ`:actx)` in round parentheses, which indicates that it is implicit and recovered during reconstruction. Variables quantified in curly braces such as `{`ϕ`:actx}` are passed explicitly. For convenience, we quantify explicitly over the terms `M1` and `M2` in the definition of `LogArr`. As in LF specifications, all free variables occurring in constructor definitions are reconstructed and bound implicitly at the outside. They are passed implicitly and recovered during reconstruction.

Crucially, logical equality is monotonic under path substitutions.

```
monotone: {ϕ:actx}{π:[ϕ ⊢ ψ]} Log [ψ ⊢ M1 ≈ M2] [A] → Log [ϕ ⊢ M1[π] ≈ M2[π]] [A]
```

We show below the mechanized proof of this lemma. The proof is simply by case analysis on the logical equivalence. In the base case, we obtain a proof P of $\psi \vdash$ `algeq M N i`, which we can weaken for free by simply applying π to P. Here we benefit significantly from Beluga's built-in support for simultaneous substitutions; we gain not just weakening by a single variable for free as we would in Twelf, but arbitrary simultaneous weakening. The proof proceeds in the arrow case by simply composing the two substitutions. We use Λ as the introduction form for universal quantifications over metavariables (contextual objects), for which we use uppercase and Greek letters, and **fn** with lowercase letters for computation-level function types (implications).

```
rec monotone:{ϕ:actx}{π:[ϕ ⊢ ψ]} Log [ψ ⊢ M1 ≈ M2] [A] → Log [ϕ ⊢ M1[π] ≈ M2[π]] [A] =
Λ ϕ,π ⇒ fn e ⇒ case e of
| LogBase [ψ ⊢ P] ⇒ LogBase [ϕ ⊢ P[π]]
| LogArr [ψ ⊢ M1] [ψ ⊢ M2] f ⇒
  LogArr (Λ ϕ',π', N1, D1, N2, D2 ⇒ fn rn ⇒
            f [ϕ'] [ϕ' ⊢ π[π']] [ϕ' ⊢ N1] [ϕ' ⊢ D1] [ϕ' ⊢ N2] [ϕ'⊢ D2] rn)
```

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. Crucially, in the arrow case, `main2` instantiates the path substitution π with a weakening substitution in order to create a fresh variable.

```
rec main1 : {A:[tp]} [ψ ⊢ algeqNeu M1 M2 A]  → Log [ψ ⊢ M1 ≈ M2] [A]
and main2   : {A:[tp]} Log [ψ ⊢ M1 ≈ M2] [A] → [ψ ⊢ algeq M1 M2 A]
```

We can state weak head closure directly as follows. The proof is structurally recursive on the type, which is implicit.

```
rec closed : [ψ ⊢ mstep N1 M1] → [ψ ⊢ mstep N2 M2] → Log [ψ ⊢ M1 ≈ M2] [T]
           → Log [ψ ⊢ N1 ≈ N2] [T]
```

3.6. *Encoding the Fundamental Theorem as a Total Function in Beluga*

The fundamental theorem requires us to speak of all instantiations of open terms by related substitutions. We express here the notion of related substitutions using inductive types.

```
inductive LogSub : {ψ:ctx}(φ:actx){σ1:[φ ⊢ ψ]}{σ2:[φ ⊢ ψ]} ctype =
| Nil : LogSub [] [φ ⊢ ·] [φ ⊢ ·]
| Dot : LogSub [ψ] [φ ⊢ σ1] [φ ⊢ σ2] → Log [φ ⊢ M1 ≈ M2] [T]
      → {D1:[φ ⊢ oft M1 R[]]}{D2:[φ ⊢ oft M2 R[]]}
        LogSub [ψ, b:block x:tm, t_x: oft x R[]] [φ ⊢ σ1, <M1;D1>] [φ ⊢ σ2, <M2;D2>]
```

We have a monotonicity lemma for logically equivalent substitutions which is similar to the monotonicity lemma for logically equivalent terms:

```
rec logsub_monotone : {φ:actx}{φ':actx}{π:[φ' ⊢ φ]}
                      LogSub [ψ] [φ ⊢ σ1] [φ ⊢ σ2]
                    → LogSub [ψ] [φ' ⊢ σ1[π]] [φ' ⊢ σ2[π]]
```

The fundamental theorem requires a proof that `M1` and `M2` are declaratively equal, together with logically related substitutions σ_1 and σ_2, and produces a proof that `M1`$[\sigma_1]$ and `M2`$[\sigma_2]$ are logically related. In the transitivity and symmetry cases, we appeal to transitivity and symmetry of logical equivalence, the proofs of which can be found in the accompanying Beluga code.

```
rec thm : [ψ ⊢ deq M1 M2 T[]] → LogSub [ψ] [φ ⊢ σ1] [φ ⊢ σ2]
        → Log [φ ⊢ M1[σ1] ≈ M2[σ2]] [T] =
```

We show the `lam` case of the proof term only to make a high-level comparison to the hand-written proof in Sec. 3. Below, one can see that we appeal to monotonicity (`logsub_monotone`), weak head closure (`closed`), and the induction hypothesis on the subderivation `d1`. However, remarkably, there is no explicit equational reasoning about substitutions, since applications of substitutions are automatically simplified. We refer the reader to [CP13] for the technical details of this simplification.

```
fn d, s ⇒ case d of

| [ψ ⊢ d_lam λx.λy. E1] ⇒
  let ([ψ ⊢ _ ] : [ψ ⊢ deq (lam λx.M1) (lam λx.M2) (arr P[] Q[])]) = d in
  LogArr [φ ⊢ lam (λx. M1[σ1[…],x])] [φ ⊢ lam (λx. M2[σ2[…],x])]
   (Λ φ',π,N1,D1,N2,D2 ⇒ fn rn ⇒
    let q0 = Dot (logsub_monotone [φ'] [φ] [φ' ⊢ π] s) rn [φ' ⊢ D1] [φ' ⊢ D2] in
    let q2 = thm [ψ,b:block x:tm,y:oft x _ ⊢ E1[…,b.1,b.2]] q0 in
    closed [⊢ Q] [φ' ⊢ trans1 beta refl] [φ' ⊢ trans1 beta refl] q2
   )
| …
```

Completeness is a corollary of the fundamental theorem.

3.7. *Remarks*

The proof passes Beluga's typechecking and totality checking. As part of the totality checker, Beluga performs a strict positivity check for inductive types [PA15, PC15], and a stratification check for logical relation-style definitions.

Beluga's built-in support for simultaneous substitutions is a big win for this proof. The proof of the monotonicity lemma is very simple, since the (simultaneous) weakening of algorithmic equality comes for free, and there is no need for explicit reasoning about substitution equations in the fundamental theorem or elsewhere. We also found that the technique of quantifying over path substitutions as opposed to quantifying over all extensions of a context to work surprisingly well.

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.

Furthermore, one might argue that having to explicitly apply the path substitutions π to terms like $M[\pi]$ is somewhat unsatisfactory, so one might wish for the ability to directly perform the bounded quantification $\forall \Phi \geq \Psi$ and a notion of subtyping which permits for example $\texttt{[}\Psi \vdash \texttt{tm]} \leq \texttt{[}\Phi \vdash \texttt{tm]}$. This is also a possibility we are exploring.

Overall, we found that that the tools provided by Beluga, especially its support for simultaneous substitutions, worked remarkably well to express this proof and to obviate the need for bureaucratic lemmas about substitutions and contexts, and we are optimistic that these techniques can scale to many other varieties of logical relations proofs.

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 implement a specification logic (similar to LF) inside a higher-order logic supporting (co)inductive definitions, the approach taken in Abella [Gac08], or type theory, the approach taken in Hybrid [FM12]. To distinguish in the proof theory between quantification over variables and quantification over terms, [GMN08] introduce a new quantifier, ∇, to describe nominal abstraction logically. To encode logical relations one uses recursive definitions which are part of the reasoning logic [GMN09]. 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 derivations. It is also not clear what is the most effective way to perform the quantification over all *extensions* of a context in Abella.

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 [PS99] 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, [SS08] proposes to implement a reasoning logic within LF and then use it to encode logical relation arguments. This approach scales to richer calculi [RF13] and avoids reasoning about contexts, open terms and simultaneous substitutions explicitly. Exploiting the same idea of an assertion logic, [**?**] outline a normalization proof implemented in an extension of Twelf that allows users to define morphisms between different signatures therby obtaining the the basic lemma "for free". 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, [NU08] have carried out essentially the same proof in Nominal Isabelle, and later [UCB11] 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

Both our case studies of Kripke-style logical relations proofs take advantage of key infrastructure provided by Beluga: it takes advantage of specifying lambda-terms together with their relevant theory in the logical framework LF, and more importantly it utilizes first-class simultaneous substitutions, contexts, contextual objects and the power of recursive types. This yields a direct and compact implementation of all the necessary proofs which directly correspond to their on-paper developments and yields an executable program. We believe these case studies demonstrate that Beluga provides the right level of abstractions and primitives to mechanize directly challenging problems such as proofs by logical relations using higher-order abstract syntax encodings. The programmer is able to concentrate on implementing the essential aspects of the proof spending his effort in the right place.

References

Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In Marc Bezem and Jan Friso Groote, editors, *International Conference on Typed Lambda Calculi and Applications (TLCA '93)*, volume 664 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 1993.

Stefano Berardi. Girard normalization proof in LEGO. In *Proceedings of the First Workshop on Logical Frameworks*, pages 67–78, 1990.

Catarina Coquand. A proof of normalization for simply typed lambda calculus writting in ALF. In *Informal Proceedings of Workshop on Types for Proofs and Programs*, pages 80–87. Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg Univ., 1992.

Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In *39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM Press, 2012.

Andrew Cave and Brigitte Pientka. First-class substitutions in contextual type theory. In *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, pages 15–24. ACM Press, 2013.

Karl Crary. Logical relations and a case study in equivalence checking. In Bejamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005.

Christian Doczkal and Jan Schwinghammer. Formalizing a strong normalization proof for Moggi's computational metalanguage: A case study in Isabelle/HOL-nominal. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'09)*, pages 57–63. ACM, 2009.

Amy Felty and Alberto Momigliano. Hybrid - A definitional two-level approach to reasoning with higher-order abstract syntax. *J. Autom. Reasoning*, 48(1):43–105, 2012.

Francisco Ferreira and Brigitte Pientka. Bidirectional elaboration of dependently typed languages. In *16th International Symposium on Principles and Practice of Declarative Programming (PPDP'14)*, pages 161–174. ACM, 2014.

Andrew Gacek. The Abella interactive theorem prover (system description). In *4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 154–161. Springer, 2008.

J.-Y. Girard, Y. Lafont, and P. Tayor. *Proofs and types*. Cambridge University Press, 1990.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *23rd Symposium on Logic in Computer Science*, pages 33–44. IEEE Computer Society Press, 2008.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. Reasoning in Abella about structural operational semantics specifications. In *Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008)*, volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 85 – 100. Elsevier, 2009.

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

Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6(1):61–101, 2005.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, 2008.

Julien Narboux and Christian Urban. Formalising in Nominal Isabelle Crary's completeness proof for equivalence checking. *Electr. Notes Theor. Comput. Sci.*, 196:3–18, 2008.

Brigitte Pientka and Andreas Abel. Structural recursion over contextual objects. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*. Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, 2015.

Brigitte Pientka and Andrew Cave. Inductive beluga:programming proofs (system description). In *25th International Conference on Automated Deduction (CADE-25)*. Springer, 2015.

Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173. ACM Press, 2008.

Brigitte Pientka and Joshua Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In Jürgen Giesl and Reiner Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21. Springer-Verlag, 2010.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM Press, 2008.

Brigitte Pientka. An insider's look at LF type reconstruction:Everything you (n)ever wanted to know. *Journal of Functional Programming*, 1(1–37), 2013.

Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th International Conference on Automated Deduction (CADE-16)*, Lecture Notes in Artificial Intelligence (LNAI 1632), pages 202–206. Springer, 1999.

Adam B. Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *17th European Symposium on Programming (ESOP '08)*, volume 4960, pages 93–107. Springer, 2008.

Ulrik Rasmussen and Andrzej Filinski. Structural logical relations with case analysis and equality reasoning. In *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP'13)*, pages 43–54. ACM Press, 2013.

Carsten Schürmann and Jeffrey Sarnat. Structural logical relations. In *23rd Annual Symposium on Logic in Computer Science (LICS), Pittsburgh, PA, USA*, pages 69–80. IEEE Computer Society, 2008.

William Tait. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.*, 32(2):198–212, 1967.

Christian Urban, James Cheney, and Stefan Berghofer. Mechanizing the metatheory of LF. *ACM Trans. Comput. Log.*, 12(2):15, 2011.