Math. Struct. in Comp. Science (2018), *vol.* 28, *pp.* 1606–1638. © Cambridge University Press 2018
doi:10.1017/S0960129518000154 First published online 2 August 2018

Mechanizing proofs with logical relations – Kripke-style

ANDREW CAVE and BRIGITTE PIENTKA

School of Computer Science, McGill University, Montreal, Canada
Email: acave1@cs.mcgill.ca, bpientka@cs.mcgill.ca

Received 8 January 2016; revised 9 December 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 (HOAS) thereby avoiding the need to manipulate and deal with binders, renaming and substitutions, (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 HOAS encodings. It also may serve as a valuable benchmark for other proof environments.

1. Introduction

Logical relations provide a key proof method to establish rich properties such as non-interference, normalization or program equivalence. 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 is in the relation. This latter part usually 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 normalization proofs using logical relations 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 (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 logical framework (LF) itself including the completeness of equivalence checking (Narboux and Urban 2008; Urban et al. 2011) which we will also discuss in this paper.

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 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 are beyond the logical strength supported in systems such as Twelf (Pfenning and Schürmann 1999). In this paper, we demonstrate the elegance of logical relations proofs within the proof environment Beluga (Pientka and Dunfield 2010) which is built on top of the contextual LF. In contrast to LF where the assumptions are implicit, the contextual LF extends LF with first-class contexts and first-class simultaneous substitutions supporting a rich equational theory about them (Cave and Pientka 2013; Pientka and Cave 2015). 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 (Nanevski et al. 2008).

Properties about contexts, substitutions and contextual objects such as for example logical relations can be encoded in Beluga using indexed inductive types (Cave and Pientka 2012). Inductive proofs about contexts and contextual LF objects are implemented in Beluga as dependently typed recursive functions via pattern matching (Pientka 2008, 2010; Pientka and Dunfield 2008).

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 (2005). Algorithmic equality is defined in a type-directed way. For example, 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 the following three key aspects:

1. We encode lambda-terms together with their typing rules, operational semantics, algorithmic and declarative equality using HOAS in the LF. This allows us to model binders in our object language (i.e. the simply typed lambda-calculus) using the binders in the 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.
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 and extensions, are accessible at

`https://github.com/Beluga-lang/Beluga/tree/master/examples/logrel.`

The mechanization of the weak normalization proof for the simply typed lambda-calculus using logical relations in Beluga has not been previously described. The completeness proof of algorithmic equality in Beluga has been presented in Cave and Pientka (2015). In this paper, we use both case studies to motivate and illustrate how to state logical relations on open terms and how to model context extensions in Beluga and propose these two examples as benchmark problems for proof environments.

Over the past decade, we have seen progress in developing a range of benchmark problems: POPLmark (Aydemir et al. 2005) lead the way and highlighted best practices when mechanizing standard theorems that only require proofs by structural induction on a given derivation. Similar in spirit, Felty, Momigliano and Pientka proposed a set of crafted benchmark problems that highlighted the role of reasoning about context (Felty et al. 2017, 2015; Felty and Pientka 2010). More recently, Kaiser et al. (2017) proposed the equivalence proof between System F and its single-sorted pure type system (PTS) variant $\lambda 2$ as a benchmark to compare different proof environments relying on HOAS encodings as well as using de Bruijn encodings. The benchmark of logical relations on open terms which we describe in this paper goes beyond these existing benchmarks as we reason semantically about reducibility. It requires us to reason about open terms, context extensions, substitutions and relies on both syntactic and semantic definitions. We would welcome and encourage others in the community to mechanize the proposed logical relations proof to initiate a dialogue on the trade-offs between different techniques.

2. Example: Weak normalization

We begin by discussing the weak normalization proof for 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 **i** and function types.

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

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 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

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

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

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

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

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 $\quad \boxed{M\ \mathsf{neut}}$ M is neutral

$$\frac{M\ \mathsf{neut}}{M\ \mathsf{norm}}\ \texttt{n\_neut} \qquad \frac{M\ \mathsf{neut} \quad N\ \mathsf{norm}}{M\ N\ \mathsf{neut}}\ \texttt{n\_app} \qquad \frac{\begin{array}{c}\overline{x\ \mathsf{neut}}\ n\\ \vdots\\ M\ \mathsf{norm}\end{array}}{\mathsf{lam}\ x.M\ \mathsf{norm}}\ \texttt{n\_lam}^{x,n}$$

We have chosen here a standard set of reduction rules which is non-deterministic, but reduction of well-typed terms is guaranteed to terminate and produce a unique normal form. There are different alternative formulations of reductions and normal forms one might consider. For example, we might only characterize weak head normal forms by stating that $M\ N$ is neutral if M is neutral and consider fixing a call-by-value strategy by removing the evaluation rule s_app2. Alternatively, we could fix a call-by-name strategy by enforcing that we only use s_β rule, if the argument is a value. These variations do not change substantially the overall proof of normalization which we show in the next section.

2.2. *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. While it is possible to define reducibility for simply

$$
\begin{array}{lcl}
\boxed{(\Psi \vdash M) \in \mathcal{R}_A} & \multicolumn{2}{l}{\text{Term } M \text{ is reducible at type } A} \\
\mathcal{R}_{\mathbf{i}} & = & \{\Psi \vdash M \mid \Psi \vdash M \;\; \text{halts}\} \\
\mathcal{R}_{A \to B} & = & \{\Psi \vdash M \mid \Psi \vdash M \text{ halts and } \forall \Phi \geq_\rho \Psi, N \text{ where } \Phi \vdash N : A, \\
 & & \quad \text{if } (\Phi \vdash N) \in \mathcal{R}_A \text{ then } (\Phi \vdash M[\rho]\; N) \in \mathcal{R}_B\}
\end{array}
$$

Fig. 1. Reducibility for well-scoped and well-typed terms.

typed terms without necessarily ensuring that reducible terms are well-scoped and well-typed (see, for example, Altenkirch (1993)), this approach does not scale to systems that rely on type-directed evaluation or equivalence checking. In particular, we will consider such a system in Section 3.

Starting from our desire to concentrate only on well-scoped and well-typed terms, this leads to the question what context(s) should we consider. Previously, we have chosen a two-dimensional representation of rules defining typing, reduction and normal forms. This representation left the context of assumptions implicit sidestepping this question. Revisiting the judgements for typing and normal forms, we note that each relies on different assumptions. Typing judgements refer to typing assumptions that we can make explicit using a typing context $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$. For characterizing normal and neutral terms, we rely on assumptions that characterize variables as neutral.

Reducibility makes statements about well-typed terms. 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 norm requires that for each variable occurring in M, we have an assumption stating that it is neutral. Hence, when we define reducibility of terms we need to consider the term M in a context that has both typing and neutral assumptions. We refer to such a context as the typed neutral variable context.

Typing Context	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Neutral Variable Context	Ω	$::=$	$\cdot \mid \Omega, x\ \mathsf{neut}$
Typed Neutral Variable Context	Ψ, Φ	$::=$	$\cdot \mid \Psi, x : A, x\ \mathsf{neut}$

We can always weaken any typing context Γ to a typed neutral variable context Ψ; similarly, any neutral variable context Ω can be weakened to a context Ψ. Moreover, because our definition of typing and normal forms were defined independently from each other, we can also strengthen any typed neutral variable context to a typing context or a neutral variable context, respectively. This may seem trivial and obvious on-paper, however, this may need to be taken into account in a mechanization. For example, in a mechanization where we model variables and their scope using de Bruijn indices, we may need to shift indices such that they remain meaningful.

We will define reducibility for a term M in the typed neutral variable context Ψ (written as $\Psi \vdash M$) to capture precisely the relevant assumptions (see Figure 1). A term M is reducible at base type, when it halts, i.e. there exists a term V it steps to and V norm in the context Ψ. To emphasize that we are working with well-scoped terms and well-typed terms, we make the typing context explicit in the stepping relation writing $\Psi \vdash M \longrightarrow M'$ and $\Psi \vdash M \longrightarrow^* M'$, respectively.

$$\frac{\Psi \vdash M \longrightarrow^* V \quad \Psi \vdash V \;\mathsf{norm}}{\Psi \vdash M \;\mathsf{halts}}$$

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 $\Phi \vdash \rho : \Psi$ (i.e. ρ has domain Ψ and co-domain Φ) and for every reducible term N at type A in the future context Φ, the application $M[\rho]\; N$ is reducible at type B in the future context Φ.

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

$\boxed{\Phi \geqslant_\rho \Psi}$ ρ is a neutral substitution from the context Ψ to Φ

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

There are two special cases of neutral substitutions that often arise: (1) *identity substitution* written as id_Ψ to denote a neutral substitution $\Psi \vdash \mathsf{id}_\Psi : \Psi$, where the range and domain are the same and any variable x declared in Ψ is mapped to itself; (2) *weakening substitution* written as wk_Ψ to denote a neutral substitution $\Psi, \Psi' \vdash \mathsf{wk}_\Psi : \Psi$, where the range of the substitution is extended and any variable in Ψ is mapped to itself.

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 judgements and when referring to our judgemental definition of normal and neutral terms. Obviously, the two-dimensional formulation which we used in Section 2.1 and the one with explicit contexts are equivalent and it is easy to convert between them.

We begin by stating two main properties about the reducibility candidates: (1) if a term M is reducible, then its evaluation is guaranteed to halt; (2) all neutral terms that are well-typed are included in the reducibility relation.

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 $\Psi \vdash M$ 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 $\Psi \vdash M \longrightarrow M'$, then $(\Psi \vdash M) \in \mathcal{R}_A$.
2. If $(\Psi \vdash M') \in \mathcal{R}_A$ and $\Psi \vdash M \longrightarrow^* M'$, 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 $\mathcal{R}_\mathbf{i}$
$\Psi \vdash M' \longrightarrow^* V$ and $\Psi \vdash V$ norm	by definition of halts
$\Psi \vdash M \longrightarrow^* V$	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. $\Psi \vdash M' \longrightarrow^* V$ and $\Psi \vdash V$ norm for some V	
for all $\Phi \geqslant_\rho \Psi$, if $\Phi \vdash N : B$ and $(\Phi \vdash N) \in \mathcal{R}_B$ then $(\Phi \vdash M'[\rho]\ N) \in \mathcal{R}_C$	by red. def.
Assume $\Phi \geqslant_\rho \Psi$, $\Phi \vdash N : B$ and $(\Phi \vdash N) \in \mathcal{R}_B$	
$\Psi \vdash M \longrightarrow M'$	by assumption
$\Psi \vdash M \longrightarrow^* V$	by `s_trans`
$\Psi \vdash M$ halts	by def. of halts
$\Phi \vdash M[\rho] \longrightarrow M'[\rho]$	by *subst. property*
$\Phi \vdash M[\rho]\ \ N \longrightarrow M'[\rho]\ \ N$	by rule `s_app1`
$(\Phi \vdash M'[\rho]\ \ N) \in \mathcal{R}_C$	by previous lines
$(\Phi \vdash M[\rho]\ \ N) \in \mathcal{R}_C$	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 $(\Phi \vdash M[\sigma]) \in \mathcal{R}_A$. 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 (i.e. the variables being replaced by the substitution) and range (i.e. the context in which the instantiations for the variables are meaningful) 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 $\Phi = y_1{:}B_1,\ y_1\ \mathsf{neut},\ \ldots, y_k{:}B_k,\ y_k\ \mathsf{neut}$, 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. $\Phi \vdash \sigma : \Psi$, i.e. σ provides a mapping from the variables in Ψ to the context Φ. 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 $\Phi \geqslant_\rho \Psi$, then $(\Phi \vdash M[\rho]) \in \mathcal{R}_A$.
2. If $(\Psi \vdash \sigma) \in \mathcal{R}_{\Psi_0}$ and $\Phi \geqslant_\rho \Psi$, then $(\Phi \vdash \sigma[\rho]) \in \mathcal{R}_{\Psi_0}$.

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 $(\Psi \vdash M[\sigma]) \in \mathcal{R}_A$.

Proof. By induction on the typing derivation, we show only the interesting cases:

Case $\dfrac{x{:}A \in \Gamma}{\Gamma \vdash x : A}$

$\Gamma = \Gamma_1, x{:}A, \Gamma_2$ — since $x{:}A \in \Gamma$
$(\Psi \vdash \sigma) \in \mathcal{R}_\Gamma$ — by assumption
$(\Psi \vdash \sigma_1) \in \mathcal{R}_{\Gamma_1}$ and $(\Psi \vdash N) \in \mathcal{R}_A$ and $\sigma = \sigma_1, N/x, \sigma_2$ — by def. of $\mathcal{R}_\Gamma$
$(\Psi \vdash x[\sigma]) \in \mathcal{R}_A$ — by *subst. property*

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

$(\Psi \vdash M[\sigma]) \in \mathcal{R}_{A\to B}$ and $(\Psi \vdash N[\sigma]) \in \mathcal{R}_A$ — by i.h.
$\forall \Phi \geqslant_\rho \Psi, N'$ where $\Phi \vdash N' : A$
if $(\Phi \vdash N') \in \mathcal{R}_A$, then $(\Phi \vdash ((M[\sigma])[\rho])\ \ N') \in \mathcal{R}_B$ — by reducibility def.
$(\Psi \vdash (M[\sigma])[\mathsf{id}_\Psi])\ \ (N[\sigma]) \in \mathcal{R}_B$ — choosing id_Ψ for ρ, Ψ for Φ, and N for N'

$(\Psi \vdash (M[\sigma])[\mathsf{id}_\Psi])\ \ (N[\sigma]) = (M[\sigma])\ (N[\sigma]) = (M\ \ N)[\sigma]$	by *subst. properties*
$(\Psi \vdash (M\ \ N)[\sigma]) \in \mathcal{R}_B$	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
$\Psi, x{:}A, x\ \mathsf{neut} \vdash \mathsf{wk}_\Psi : \Psi$ and $\Psi, x{:}A, x\ \mathsf{neut} \geqslant_{\mathsf{wk}_\Psi} \Psi$	by *weakening*
$(\Psi, x{:}A, x\ \mathsf{neut} \vdash \sigma[\mathsf{wk}_\Psi]) \in \mathcal{R}_\Gamma$	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)
$(\Psi, x{:}A, x\ \mathsf{neut} \vdash \sigma[\mathsf{wk}_\Psi],\ x/x) \in \mathcal{R}_{\Gamma, x:A}$	by def. of reducibility of substitutions
$(\Psi, x{:}A, x\ \mathsf{neut} \vdash M[\sigma[\mathsf{wk}_\Psi],\ x/x]) \in \mathcal{R}_B$	by i.h.
$\Psi, x{:}A, x\ \mathsf{neut} \vdash M[\sigma[\mathsf{wk}_\Psi],\ x/x]$ halts	by Main Lemma (1)
$\Psi, x{:}A, x\ \mathsf{neut} \vdash M[\sigma[\mathsf{wk}_\Psi],\ x/x] \longrightarrow^* V$ and $\Psi, x{:}A, x\ \mathsf{neut} \vdash V\ \mathsf{norm}$	by def. of halts
$\Psi \vdash \mathsf{lam}\ x.M[\sigma[\mathsf{wk}_\Psi],\ x/x] \longrightarrow^* \mathsf{lam}\ x.V$	by multi-step red. for lambda-abstractions
$\Psi \vdash (\mathsf{lam}\ x.M)[\sigma] \longrightarrow^* \mathsf{lam}\ x.V$	by *subst. property*
$\Psi \vdash \mathsf{lam}\ x.V\ \mathsf{norm}$	by `n_lam`
$\Psi \vdash (\mathsf{lam}\ x.M)[\sigma]$ halts	by def. of halts

Given an extension $\Phi \geqslant_\rho \Psi$ and a term N, s.t. $\Phi \vdash N : A$ and $(\Phi \vdash N) \in \mathcal{R}_A$.

$(\Phi \vdash \sigma[\rho]) \in \mathcal{R}_\Gamma$	by monotonicity lemma
$(\Phi \vdash (\sigma[\rho],\ N/x)) \in \mathcal{R}_{\Gamma, x:A}$	by def. of reducibility for substitutions
$(\Phi \vdash M[\sigma[\rho],\ N/x]) \in \mathcal{R}_B$	by i.h.
$(\Phi \vdash (M[\sigma[\rho],\ x/x])[N/x]) \in \mathcal{R}_B$	by *subst. property*
$(\Phi \vdash (\mathsf{lam}\ x.M[\sigma[\rho],\ x/x])\ \ N) \in R_B$	by backwards closed lemma
$(\Phi \vdash (\mathsf{lam}\ x.M)[\sigma[\rho]]\ \ N) \in R_B$	by *subst. properties*
$(\Phi \vdash ((\mathsf{lam}\ x.M)[\sigma])[\rho]\ \ N) \in R_B$	by *subst. properties*
$(\Psi \vdash (\mathsf{lam}\ x.M)[\sigma]) \in R_{A \to B}$	by def. of reducibility

□

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 $(\Phi \vdash M[\mathsf{id}]) \in \mathcal{R}_A$ and hence by reification $\Phi \vdash M[\mathsf{id}]$ halts. By *subst. property*, we have $M[\mathsf{id}] = M$ 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 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. At the same time, our definitions and use of context extensions are natural and commonly used in reasoning about dependently typed calculi (see, for example, the work by Abel and Scherer (2012) or the mechanization of dependent type theory using logical relations by Öhmann (2016)). By using the same methodology for reasoning about simply typed systems, we can study the role of context extensions more clearly while at the same time providing a stepping stone towards establishing normalization properties about dependently typed calculi.

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 HOAS 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. We can then encode lam x.(lam y.(y x)) as `lam λx. lam λy. app y x`. When declaring the type of each of the constant `app` and `lam`, the indices `A` and `B` are free. Type reconstruction will infer the type of free variables and abstract over them at the outside, but they are marked as implicit. As a consequence, we do not pass instantiations for them when building terms, but type reconstruction will infer them (Pientka 2013).

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

```
LF tm : tp → type =
| app : tm (arr A B) → tm A → tm B
| lam : (tm A → tm B) → tm (arr A B);
```

Next, we 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. Further, in the definition of `s_lam`, we declare that `(lam M)` steps to `(lam M')`, if for all variables `x`, we have that `M x` steps to `M' x`. We rename here the bound variable in `M` and `M'` simply by applying each term to a fresh variable `x`. Hence, we again fall back to LF application to model renaming. Finally, we point out that we write curly braces `{ }` for universal quantification in Beluga.

```
LF step   : tm A → tm A → type =
| s_beta : step (app (lam M) N) (M N)
| s_lam  : ({x:tm A} step (M x) (M' x)) → step (lam M) (lam M')
| 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 judgements 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 M)`. Note that `M` has type `tm A → tm B`. A term `lam M` is normal, if we can show that for all `x:tm A` 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 M)
| n_neut : neutral M → normal M
and neutral : tm A → type =
| n_app : neutral M → normal N → neutral (app M N);
```

For a more detailed explanation on how to encode formal systems using HOAS, we refer the reader to the tutorial (Pientka 2015) which provides a gentle introduction to encoding formal systems in LF and Beluga. For a guide to the Beluga surface language, we refer the reader to the appendix A.

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 LF, the context of assumptions is ambient and implicit. This makes encoding of logical relations in LF challenging. Luckily, Beluga not only supports LF specifications, but provides a first-order logic with inductive definitions (Cave and Pientka 2012) 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 (Nanevski et al. 2008). 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 and sometimes even essential to abstract over the concrete context of assumptions to express meaningful properties. In Beluga, we can classify contexts using context schemas. Schema declarations in Beluga express an invariant we want to hold when stating inductive properties about LF objects. The schema declaration `ctx` below states that a context of schema `ctx` contains assumptions that are instances of `tm A`. We use the **some** keyword to abstract over variables in a schema that are fexible, i.e. they can be instantiated. 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 typing context and typed neutral variable context from page 1611 forcing the fact that the typing assumption `x:tm A` is introduced together with the assumption `neut x` which states that the variable `x` is neutral. Formally, a block allows us to pair several assumptions together using a Σ-type.

```
schema ctx  = some [A:tp] tm A;
schema nctx = some [A:tp] block x:tm A, 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 $\Phi \geqslant_\rho \Psi$ 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 1611 into an indexed recursive type.

Beluga supports inductive and stratified definitions. Inductive definitions, introduced with the keyword **inductive**, satisfy the positivity restriction and correspond to least fixed points. Moreover, we can reason about them by induction. On the other hand, stratified definitions, introduced with the keyword **stratified** are defined recursively over an index; they do not need to satisfy the positivity restriction. To reason about stratified definitions, we reason by induction on the index directly. 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[]]} type =
| 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[]]} type =
| 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, whereas 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 `Ψ ⊢ 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`. A stratified type is 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

```
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));
```

Fig. 2. Encoding of backwards closed lemma in Beluga.

contexts Φ, substitutions ρ from Ψ to Φ, and terms `N` of type `A` in the context Φ that are reducible, we have that `[Φ ⊢ 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 (Pfenning and Schürmann 1999), Beluga's type reconstruction infers types for any free variable in a given type or kind declaration and implicitly quantifies over them (Ferreira and Pientka 2014; Pientka 2013). Programmers subsequently do not need to supply arguments for implicitly quantified variables.

We refer the reader to the Appendix A for a description of Beluga's grammar and intuition behind the definitions.

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) informally below, such that it 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:[Ψ ⊢ step M M']`, if `Reduce [A] [Ψ ⊢ M']`, then `Reduce [A] [Ψ ⊢ M]`.

We describe the implementation of the lemma in Figure 2. As explained in the Appendix A, Beluga's computation language distinguishes between dependent and simple function types. We use Λ-abstractions for the former and **fn**-abstractions for the latter. As the lemma was proven by induction on the type of the term, our corresponding function `closed` proceeds by pattern matching on the type `A`. In the base case where `A`=`i`, we model inversion in the actual derivation `Reduce [i] [Ψ ⊢ M']` using pattern matching on the definition of `Reduce` and subsequently on the definition of `Halt` obtaining a derivation `S'` for `[Ψ ⊢ step M' V]` and a proof `v` that `V` is normal. We then use the assumption `S` which stands for `[Ψ ⊢ step M M']` and `S'`, and build a derivation for `[Ψ ⊢ step M V]` using transitivity (i.e. the rule `s_trans`). From this, it is now trivial to provide a witness that `Reduce [i] [Ψ ⊢ M]`.

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

We now build a witness for `Reduce [C] [Φ ⊢ 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] [Φ ⊢ N]` and returns `Reduce [C] [Φ ⊢ app M[ρ] N]` following our earlier proof arguing by induction (recursion).

The Beluga directive `/ total A (closed _ A) /` checks that that the function `closed` is total, i.e. there exists a case for all possible input (coverage) and all recursive calls are on smaller types `A` (termination) (Pientka and Abel 2015). In specifying the total directive, the position of the input argument indicates the recursive argument. In this example, the function `closed` takes in first (implicitly) a context Ψ followed by a type `A`. Hence, the function 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 main lemma follows the on-paper proof directly. 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.

```
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]
```

Now we 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 `[Φ ⊢ Γ]`, 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, written as `[Φ ⊢ ␣ ]`, is reducible. In the `Cons` case, we read this as saying: if σ is a reducible substitution (implicitly at type `[Φ ⊢ Γ]`) and `M` is a reducible term at type `A`, then σ with `M` appended is a reducible substitution (implicitly at type `[Φ ⊢ Γ,x:tm A[]]` – the domain has been extended with a variable of type `A`).

```
inductive LogSub : {Γ:ctx}(Φ:nctx){σ:[Φ ⊢ Γ]} type =
| 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 also explicitly quantify over substitution variables using curly braces writing `{σ:[Φ ⊢ Γ]}`. 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.

```
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. M] ⇒
   let rx = main2 [Φ,b:block x:tm _,y:neutral x] [_] [Φ,b ⊢b.1] [Φ,b ⊢b.2] in
   let q0 = fund [Γ,x:tm _ ⊢ M]
                (Dot (monotone_sub [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]
     = main1 [Φ,b:block x:tm _,y:neutral x] [_] [Φ,b ⊢ M[σ[…],b.1]] q0 in
   Arr [Φ ⊢ lam (λx.M[σ[…], x])]
   (Halts [_ ⊢ _]
     (m_lam [Φ, x:tm A1[] ⊢ M[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.M[σ[ρ[…]],x]) N ] [Ψ ⊢ M[σ[ρ], N] ]
              [Ψ ⊢ s_beta]
              (fund [Γ,x:tm _ ⊢ M] (Dot (monotone_sub [Ψ ⊢ ρ] rs) rN)))
```

Fig. 3. Implementation of the fundamental lemma in Beluga.

1. Given a neutral substitution σ with domain Ψ and range Φ, if `Reduce [A] [Ψ ⊢ M]`, then `Reduce [A] [Φ ⊢ M[σ]]`

```
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.

```
monotone_sub: {ρ:[Φ ⊢ Ψ]} LogSub [Γ] [Ψ ⊢ σ] → LogSub [Γ] [Φ ⊢ σ[ρ]]
```

Finally, our main lemma (see Figure 3) 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 (written as `Γ ⊢ #p`), we appeal to `redvar_monotone`, which is a special case of monotonicity lemma `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. We remark that underscores allows users to omit writing certain arguments explicitly and type reconstruction fill them in instead. Further, we write .. for both the identity and weakening substitution; in particular, .. is a weakening substitution from the context Φ to Φ, `x:tm _`. It is also the witness that allows us to move from Φ to Φ, `b:`**`block`**` (x:tm A1[], y:neutral x)`.

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.

Intuitively, when checking that two LF objects are equal, Beluga not only normalizes LF objects relying on $\beta\eta$-reductions but also takes into account the equational theory of simultaneous substitutions. The full equational theory is presented in Cave and Pientka (2013). We show here a few concrete instances that arise in this example to illustrate.

Congruence	`(λx.M)[σ]`	`=λx.M[σ, x]`
	`(app M N)[σ]`	`=app M[σ] N[σ]`
	`(lam λx.M)[σ]`	`=lam (λx.M)[σ]`
	`(σ, N)[σ']`	`=σ[σ'], N[σ']`
Associativity	`(M[σ])[σ']`	`=M[σ[σ']]`
	`(σ[σ'])[σ'']`	`=σ[σ'[σ'']]`
Decomposition	`(M[σ, N])`	`=(M[σ, x])[..,N]`

Weak normalization is now a trivial corollary. We only need to show that given a term the context Γ there always exists 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.

Conceptually, our approach may be most closely related to modelling terms using well-scoped de Bruijn indices (Benton et al. 2012; Öhmann 2016). However, our foundation alleviates the user from manipulating de Bruijn indices and de Bruijn substitutions directly exploiting the abstractions provided by using contextual LF. This leads to a more compact mechanization. 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 meta-theory of dependently typed systems such as LF and varieties of Martin-Löf type theory, where they are used to establish decidability of type checking.

$\boxed{M \equiv N : A}$ 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. 4. Declarative equivalence.

3.1. *The basics: Declarative and algorithmic equality*

We revisit again the simply typed lambda-calculus from the previous section. However, here we 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 Crary (2005) and Harper and Pfenning (2005).

Declarative equivalence (see Figure 4) 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:

$$\frac{M : \mathrm{Unit}\ \ N : \mathrm{Unit}}{M \equiv N : \mathrm{Unit}}$$

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 Figure 5).

3.2. *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

$\boxed{M \Leftrightarrow N : A}$ 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

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

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

Fig. 5. Algorithmic equivalence.

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. $x \leftrightarrow x : A$.

Typing context	$\Gamma \ ::= \ \cdot \mid \Gamma,\ x : A$
Well-typed algorithmic equality context	$\Psi \ ::= \ \cdot \mid \Psi,\ x : A,\ n_x : (x \leftrightarrow x{:}A)$

This context is again the generalization of the typing context and the pure algorithmic equality context. 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

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

The key case is at function type. We directly define logically equivalent terms at function type as taking logically equivalent arguments to logically equivalent results. Algorithmic equality for terms M and N of type $A \rightarrow B$ 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 be avoided as we are not forced to define reducibility on well-typed terms, it is necessary and naturally arises in the completeness proof of algorithmic equality where equality is defined in a type-directed way.

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.*

We define the abbreviation $\Phi \geqslant_\pi \Psi$ for a path substitution, i.e. a substitution that only replaces variables in Ψ by neutral terms (or paths) of the expected type.

$$\boxed{\Phi \geqslant_\pi \Psi} \quad \pi \text{ is a path substitution from the context } \Psi \text{ to } \Phi$$

$$\frac{}{\Phi \geqslant_{\cdot} \cdot} \qquad \frac{\Phi \geqslant_\rho \Psi \quad \Phi \vdash M : A \quad \Phi \vdash M \leftrightarrow M : A}{\Phi \geqslant_{\rho, M/x} \Psi, x : A,\ n_x : (x \leftrightarrow x{:}A)}$$

Our definition of logically equivalent terms generalizes Crary's definition by witnessing the context extension by path substitutions.

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

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.

As in the previous weak normalization proof for simply typed lambda-terms, a weakening substitution that would provide variables not neutral terms (or paths) would suffice. However, working with path substitutions is convenient in our setting.

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

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 Crary (2005).

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 $M_1/x_1, \ldots, M_n/x_n$ and σ_2 is of the form $N_1/x_1, \ldots, N_n/x_n$ and Γ is of the form $x_1{:}A_1, \ldots, x_n{:}A_n$, 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.

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

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 $\Delta' \geqslant_\pi \Delta$ 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 $\Delta \geqslant \Gamma$. However, we would argue that reasoning instead about extensions $\Delta \geqslant \Gamma$ 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 HOAS. 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 an inductive definition*

To define logical equivalence, we need the notion of path substitution mentioned in Section 3.2. For this purpose, we use again Beluga's built-in notion of simultaneous substitutions. We write `[Φ ⊢ Ψ]` 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 ≈ simply constructs a pair of terms:

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

Logical equivalence, written `Log [Ψ ⊢ 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 (Jacob-Rao et al. 2018). 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] → type =
| 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 Section 3: `M1` is related to `M2` in Ψ if, for any extension $\Phi \geqslant_\pi \Psi$, 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.

Crucially, logical equality is monotonic under path substitutions (see Lemma 3.1). The mathematical definition of Lemma 3.1 can be directly encoded using Beluga's first-class support for contexts, substitutions and contextual objects.

```
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 by case analysis on the logical equivalence. In the base case, we obtain a proof P of `[Ψ ⊢ 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.

```
rec monotone:{Φ:actx}{π:[Φ ⊢ Ψ]}Log [Ψ ⊢ M1 ≈ M2] [A] → Log [Φ ⊢ M1[π] ≈ M2[π]] [A] =
ΛΦ,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)
```

We can state weak head closure directly by translating the Lemma 3.2. The proof is structurally recursive on the type `T`.

```
rec closed : {A:[tp]}[Ψ ⊢ mstep N1 M1] → [Ψ ⊢ mstep N2 M2]
          → Log [Ψ ⊢ M1 ≈ M2] [A] → Log [Ψ ⊢ N1 ≈ N2] [A]
```

Last, we translate the main lemma 3.3 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]
```

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. Note that we relate substitutions that map typing context ψ to well-typed algorithmic equality context ϕ silently incorporating weakening. Trivially, empty substitutions, written as `_`, are related at empty domain. If σ_1 and σ_2 are related at ψ, `M1` and `M2` are related at `A`, and `D1` (and resp. `D2`) describe the derivation that `M1` (resp. `M2`) is well-typed, then σ_1`,<M1;D1>` and σ_2`,<M2;D2>` are related at `Ψ, b:`**block** `x:tm, t_x: oft x R[]`. Note that we pair `<M1;D1>` (and resp. `<M2;D2>`) to provide witness for the **block** `x:tm, t_x: oft x R[]`.

```
inductive LogSub : {Ψ:ctx}(Φ:actx){σ1:[Φ ⊢ Ψ]}{σ2:[Φ ⊢ Ψ]} type =
| 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 monotone_sub : {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 handwritten proof in Section 3. Below, one can see that we appeal to monotonicity (`monotone_sub`), 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 Cave and Pientka (2013) 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 (monotone_sub [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 type checking and totality checking. As part of the totality checker, Beluga performs a strict positivity check for inductive types (Pientka and Abel 2015; Pientka and Cave 2015), and a stratification check for logical relation-style definitions, i.e. it verifies that the definition is defined inductively over one of its indices (Jacob-Rao et al. 2018).

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 works surprisingly well. This could also prove us useful in mechanizations that use well-scoped de Bruijn encodings to mechanize formal systems and proofs.

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 $\forall\Phi \geqslant \Psi$ and simply allow using a term `tm` in a context ψ also in a context ϕ. During an elaboration phase, we then can insert weakening substitutions in the appropriate places to convert terms. 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 the following 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 (Gacek 2008), or inside a type theory, the approach taken in Hybrid (Felty and Momigliano 2012). To distinguish in the proof theory between quantification over variables and quantification over terms, Gacek et al. (2008) introduce a new quantifier, ∇, 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 derivations. It is also not clear what is the most effective way to perform the quantification over all *extensions* of a context in Abella that is needed to model logical relations on open terms.

(2) The type-theoretic approaches fall into two categories: we either remain within the LF 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) propose

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. Exploiting the same idea of an assertion logic, Rabe and Sojakova (2013) show how to use an extension of Twelf that allows users to define morphisms between different signatures to express proofs by logical relation. However, one might argue that this approach 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 to describe logical relations on open terms due to the need to talk about context extensions.

Outside the world of HOAS, Narboux and Urban (2008) have carried out essentially the same proof of algorithmic completeness 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, and 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 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 HOAS encodings. The programmer is able to concentrate on implementing the essential aspects of the proof spending his effort in the right place.

There are two natural extensions we plan to explore in depth in the future: First, we aim to explore mechanizing not only inductive, but also coinductive proofs. This is particularly important, when we reason about contextual equivalence: m and n are equivalent when, if inserted in *any* larger program fragment (context), both larger programs evaluate to the same value, or equivalently both terminate. While this notion of program equivalence is intuitive, it is often difficult to reason about it, mainly due to the quantification on every possible context. *Bisimilarity* has emerged as a more manageable, yet, in this setting, equivalent idea. Roughly, m and n are *bisimilar* if whenever m evaluates to a value, so does n, and all the subprograms of the resulting values are also bisimilar, and vice versa. To show congruence properties of (coinductively defined) (bi)simulation, it does not suffice to consider only closed terms, but we rely on open simulation as a relation on open well-typed terms, i.e. terms that are well-typed within a context. In addition, proving directly the congruence properties will often fail – in this case, Howe's method (Pitts 1997), a remarkably flexible and powerful syntactic technique, comes to the rescue.

The central idea is to define a Howe-relation that logically relates open terms and show that when terms are Howe-related then they are (bi)similar. Defining and reasoning about open terms and relations on them is central in this method.

We have recently extended Beluga to support coinductive definitions (Thibodeau et al. 2016) and are exploring the mechanization of Howe's method for a simply typed lambda-calculus with recursion and lazy lists (Pitts 1997). Our early results confirm that Beluga's support for modelling relations on open terms together with its built-in support for simultaneous substitution also apply to this setting and lead to a compact and elegant mechanization. In the future, we plan to explore in more depth case studies that involve both inductive and coinductive reasoning on open terms.

Second, we aim to explore mechanizing richer properties, such as strong normalization, as well as richer languages, for example, normalization of System F. In particularly, the latter has not been tackled in proof assistants based on HOAS, as existing frameworks such as Beluga and Abella lack the ability to express logical relations for polymorphic types or recursive types. To express logical relations for System F or recursive types, typically requires higher ranked polymorphism (or higher order logic). For Beluga, we do not see any fundamental new challenges in supporting higher ranked polymorphism.

References

Abel, A. and Scherer, G. (2012). On irrelevance and algorithmic equality in predicative type theory. *Logical Methods in Computer Science* **8** (1) 1–36. TYPES'10 special issue.

Altenkirch, T. (1993). A formalization of the strong normalization proof for system F in LEGO. In: Bezem, M. and Groote, J. F. (eds.) *International Conference on Typed Lambda Calculi and Applications (TLCA '93)*, Lecture Notes in Computer Science, vol. 664, Springer, 13–28.

Aydemir, B., Bohannon, A., Fairbairn, M., Foster, J., Pierce, B., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S. and Zdancewic, S. (2005). Mechanized metatheory for the masses: The POPLmark challenge. In: Hurd, J. and Melham, T. F. (eds.) *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, Lecture Notes in Computer Science, vol. 3603, Springer, 50–65.

Benton, N., Hur, C., Kennedy, A. and McBride, C. (2012). Strongly typed term representations in coq. *Journal of Automated Reasoning* **49** (2) 141–159.

Berardi, S. (1990). Girard normalization proof in LEGO. In: *Proceedings of the 1st Workshop on Logical Frameworks* 67–78.

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

Cave, A. and Pientka, B. (2013). First-class substitutions in contextual type theory. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, ACM Press, 15–24.

Cave, A. and Pientka, B. (2015). A case study on logical relations using contextual types. In: Cervesato, I. and Chaudhuri, K. (eds.) *Proceedings of the 10th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'15)*, Electronic Proceedings in Theoretical Computer Science (EPTCS), 18–33.

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

Crary, K. (2005). Logical relations and a case study in equivalence checking. In: Pierce, B. C. (ed.) *Advanced Topics in Types and Programming Languages*, The MIT Press.

Doczkal, C. and Schwinghammer, J. (2009). Formalizing a strong normalization proof for Moggi's computational metalanguage: A case study in Isabelle/HOL-nominal. In: *Proceedings of the 4th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'09)*, ACM, 57–63.

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

Felty, A.F., Momigliano, A. and Pientka, B. (2017). Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions. *Mathematical Structures in Computer Science*.

Felty, A.P., Momigliano, A. and Pientka, B. (2015). The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2 - a survey. *Journal of Automated Reasoning* **55** (4) 307–372.

Felty, A.P. and Pientka, B. (2010). Reasoning with higher-order abstract syntax and contexts: A comparison. In: Kaufmann, M. and Paulson, L. C. (eds.) *International Conference on Interactive Theorem Proving*, Lecture Notes in Computer Science, vol. 6172, Springer, 227–242.

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

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

Gacek, A., Miller, D. and Nadathur, G. (2008). Combining generic judgments with recursive definitions. In: Pfenning, F. (ed.) *Proceedings of the 23rd Symposium on Logic in Computer Science*, IEEE Computer Society Press, 33–44.

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

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

Harper, R., Honsell, F. and Plotkin, G. (1993). A framework for defining logics. *Journal of the ACM* **40** (1) 143–184.

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

Jacob-Rao, R., Pientka, B. and Thibodeau, D. (2018). Index-stratified types. In: Kirchner, H. (ed.) *Proceedings of the 3rd International Conference on Formal Structures for Computation and Deduction (FSCD)*, Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl.

Kaiser, J., Pientka, B. and Smolka, G. (2017). Relating system F and $\lambda 2$: A case study in Coq, Abella and Beluga. In: Miller, D. (ed.) *Proceedings of the 2nd International Conference on Formal Structures for Computation and Deduction (FSCD)*, Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, 21:1–21:19.

Nanevski, A., Pfenning, F. and Pientka, B. (2008). Contextual modal type theory. *ACM Transactions on Computational Logic* **9** (3) 1–49.

Narboux, J. and Urban, C. (2008). Formalising in Nominal Isabelle Crary's completeness proof for equivalence checking. *Electronic Notes in Theoretical Computer Science* **196** 3–18.

Öhmann, J. (2016). A logical relation for dependent type theory formalized in agda. Master's thesis, University of Gothenburg.

Pfenning, F. and Schürmann, C. (1999). System description: Twelf – A meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, Lecture Notes in Artificial Intelligence (LNAI), vol. 1632, Springer, 202–206.

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

Pientka, B. (2010). Beluga: Programming with dependent types, contextual data, and contexts. In: Blume, M., Kobayashi, N. and Vidal, G. (eds.) *Proceedings of the 10th International Symposium on Functional and Logic Programming (FLOPS'10)*, Lecture Notes in Computer Science (LNCS), vol. 6009, Springer, 1–12.

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

Pientka, B. (2015). *Mechanizing Types and Programming Languages: A Companion*, McGill University. Available at `https://github.com/Beluga-lang/Meta`.

Pientka, B. and Abel, A. (2015). Structural recursion over contextual objects. In: Altenkirch, T. (ed.) *Proceedings of the 13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, 273–287.

Pientka, B. and Cave, A. (2015). Inductive Beluga: Programming proofs (system description). In: Felty, A. P. and Middeldorp, A. (eds.) *Proceedings of the 25th International Conference on Automated Deduction (CADE-25)*, Lecture Notes in Computer Science (LNCS), vol. 9195, Springer, 272–281.

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

Pientka, B. and Dunfield, J. (2010). Beluga: A framework for programming and reasoning with deductive systems (system description). In: Giesl, J. and Haehnle, R. (eds.) *Proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), Springer, 15–21.

Pitts, A.M. (1997). Operationally based theories of program equivalence. In: Dybjer, P. and Pitts, A. M. (eds.) *Semantics and Logics of Computation*. New York, NY: Cambridge University Press.

Poswolsky, A.B. and Schürmann, C. (2008). Practical programming with higher-order encodings and dependent types. In: *Proceedings of the 17th European Symposium on Programming (ESOP '08)*, Lecture Notes in Computer Science (LNCS), vol. 4960, Springer, 93–107.

Rabe, F. and Sojakova, K. (2013). Logical relations for a logical framework. *ACM Transactions on Computational Logic* **14** (4) 32.

Rasmussen, U. and Filinski, A. (2013). Structural logical relations with case analysis and equality reasoning. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP'13)*, ACM Press, 43–54.

Schürmann, C. and Sarnat, J. (2008). Structural logical relations. In: *Proceedings of the 23rd Annual Symposium on Logic in Computer Science (LICS)*, Pittsburgh, PA, USA: IEEE Computer Society, 69–80.

Tait, W. (1967). Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic* **32** (2) 198–212.

Thibodeau, D., Cave, A. and Pientka, B. (2016). Indexed codata. In: Garrigue, J., Keller, G. and Sumii, E. (eds.) *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16)*, ACM, pp. 351–363.

Urban, C., Cheney, J. and Berghofer, S. (2011). Mechanizing the metatheory of LF. *ACM Transactions on Computational Logic* **12** (2) 15.

Appendix A. Overview of Beluga Source Language

A Beluga signature consists of LF declarations, (co)inductive and stratified definitions, context schema declarations and programs. We concentrate here on the inductive and stratified definitions, as these are relevant for understanding the mechanization described in the given paper and omit coinductive definitions. Similarly, we will curtail the surface grammar of Beluga and concentrate on the fragment that is in fact used in the given case studies. To describe the full surface language of Beluga is beyond the scope of this paper.

It is also important to keep in mind that Beluga's surface language is evolving; while we hope the grammar of Beluga's surface language is helpful, to understand the concise meaning of the language, we refer to reader to Cave and Pientka (2012, 2015). We describe the surface language supported as of July 2017.

For each LF type family a, we declare its LF kind together with the constants that allow us to form objects that inhabit the type family. We also support mutual LF definitions that we do not capture in our grammar below to not complicate matters.

$$
\begin{array}{lcl}
\text{Signature Decl. } \mathcal{D} & ::= & \textbf{LF}\ \mathtt{a}\ :\ \mathcal{K}_{\mathsf{LF}}\ =\ \mathtt{c}_1\ :\ \mathcal{A}_1 \mid \ldots \mid \mathtt{c}_k\ :\ \mathcal{A}_k; \\
& & \textbf{inductive}\ \mathtt{a}\ :\ \mathcal{K}\ =\ \mathtt{c}_1 : \mathcal{T}_1 \mid \ldots \mid \mathtt{c}_n : \mathcal{T}_n; \\
& & \textbf{stratified}\ \mathtt{a}\ :\ \mathcal{K}\ =\ \mathtt{c}_1 : \mathcal{T}_1 \mid \ldots \mid \mathtt{c}_n : \mathcal{T}_n; \\
& & \textbf{schema}\ \mathtt{c}\ =\ \mathcal{S}; \\
& & \textbf{rec}\ \mathtt{c}\ :\ \mathcal{T}\ =\ \mathcal{E};
\end{array}
$$

In our case study described in Section 3.3, we define for example LF kinds `tp`, `tm` as well as `step`, `mstep`, and `normal` and `neutral`. To define reducibility candidates, we use a stratified type definition; we also employ inductive definitions, for example, `Halts` characterizes when evaluation of a given term halts while `LogSub` describes reducible substitutions. Inductive definitions correspond to indexed recursive types and semantically are interpreted as least fixed points. In addition to kind and type declarations, we also support schema declarations that characterize the invariant a given LF context satisfies. In the case studies, we use several context schema declarations, namely `nctx` (well-typed neutral variable context), `ctx` (for typing contexts) and `actx` (for well-typed algorithmic equality contexts). Last, our signature contains recursive programs together with their type.

We describe Beluga's type and terms more precisely in Figure A.1. The syntax for LF kinds, types and terms is close to the syntax in the Twelf system. We write curly braces `{ }` for the dependent LF function space and allow users to write simply →, if there is no dependency. LF kinds, types and LF terms used in a signature declaration must be pure, i.e. they cannot refer to LF objects highlighted in blue. In particular, pure LF objects

LF Kinds	$\mathcal{K}_{\mathsf{LF}}$	::=	$\mathbf{type} \mid \{\mathtt{X}{:}\mathcal{A}\}\mathcal{K}_{\mathsf{LF}} \mid \mathcal{A} \rightarrow \mathcal{K}_{\mathsf{LF}}$
LF Types	$\mathcal{A}$	::=	$\mathtt{a}\ \mathcal{M}_1 \ldots \mathcal{M}_n \mid \{\mathtt{X}{:}\mathcal{A}\}\mathcal{A}' \mid \mathcal{A} \rightarrow \mathcal{A}'$
LF Terms	$\mathcal{M}$	::=	$\lambda\mathtt{x}.\mathcal{M} \mid \mathcal{H}\ \mathcal{M}_1 \ldots \mathcal{M}_n \mid \mathtt{X}[\sigma]$
LF Heads	$\mathcal{H}$	::=	$x \mid \mathtt{c} \mid \mathtt{\#p}[\sigma] \mid \mathcal{H}.\mathtt{i}$
LF Declaration	$\mathcal{D}$	::=	$\mathcal{A} \mid \mathbf{block}\ \mathtt{x}_1{:}\mathcal{A}_1,\ \ldots,\ \mathtt{x}_n{:}\mathcal{A}_n$
LF Context	$\Psi,\ \Phi$	::=	$\llcorner\!\lrcorner \mid \psi \mid \Psi,\mathtt{x}{:}\mathcal{D}$
LF Subst.	σ	::=	$\llcorner\!\lrcorner \mid \ldots \mid \sigma, \mathcal{M} \mid \sigma,\ <\mathcal{M}_1;\ \ldots;\mathcal{M}_n>$
LF Schema	$\mathcal{S}$	::=	$\mathbf{some}\ [\mathtt{y}_1{:}\mathcal{A}'_1,\ \ldots,\ \mathtt{y}_k{:}\mathcal{A}'_k]\ \mathbf{block}\ \mathtt{x}_1{:}\mathcal{A}_1,\ \ldots,\ \mathtt{x}_n{:}\mathcal{A}_n$
Contextual Type	$\mathcal{U}$	::=	$[\Psi \vdash \mathtt{a}\ \mathcal{M}_1 \ldots \mathcal{M}_n] \mid [\mathtt{c}] \mid [\Psi \vdash \Phi] \mid \ldots$
Contextual Object	$\mathcal{C}$	::=	$[\Psi \vdash \mathcal{M}] \mid [\Psi] \mid [\Psi \vdash \sigma] \mid \ldots$

Beluga Kinds	$\mathcal{K}$	::=	$\mathbf{type} \mid \{\mathtt{X}{:}\mathcal{U}\}\mathcal{K} \mid \mathcal{U} \rightarrow \mathcal{K}$
Beluga Types	$\mathcal{T}$	::=	$\{\mathtt{X}{:}\mathcal{U}\}\mathcal{T} \mid (\mathtt{X}{:}\mathtt{c})\mathcal{T} \mid \mathcal{T} \rightarrow \mathcal{T} \mid \mathcal{U} \mid \mathtt{a}\ \mathcal{C}_1 \ldots \mathcal{C}_n$
Beluga Terms	$\mathcal{E}$	::=	$\Lambda\ \vec{X} \Rightarrow \mathcal{E} \mid \mathbf{fn}\ \vec{x} \Rightarrow \mathcal{E} \mid \mathbf{case}\ \mathcal{E}\ \mathbf{of}\ \mathcal{B} \mid \mathbf{let}\ \mathcal{P} = \mathcal{E}\ \mathbf{in}\ \mathcal{E} \mid \mathcal{E}\ \mathcal{E} \mid \mathcal{C} \mid \mathtt{x} \mid \mathtt{c}$
Beluga Branches	$\mathcal{B}$	::=	$\llcorner\!\lrcorner \mid (\mathcal{B} \mid \mathcal{P} \Rightarrow \mathcal{E})$
Beluga Pattern	$\mathcal{P}$	::=	$\mathtt{x} \mid \mathcal{C} \mid \mathtt{c}\ \mathcal{P}_1 \ldots \mathcal{P}_n$

Fig. A.1. Grammar of Beluga.

cannot contain closures written as $\mathtt{X}[\sigma]$ and $\mathtt{\#p}[\sigma]$. Here, $\mathtt{X}$ and $\mathtt{\#p}$ are meta-variables that are bound and introduced in Beluga types and patterns. Furthermore, we may construct LF contexts in Beluga programs that satisfy the rich invariant that context schemas allow us to express. In particular, such LF contexts may be build out of blocks $\mathtt{x}_1{:}\mathcal{A}_1,\ \ldots,\ \mathtt{x}_n{:}\mathcal{A}_n$; formally, this denotes a Σ-type which allows us to group LF types together. We can access an element of a block using projections written in general as $\mathcal{H}.i$.

LF contexts may be empty, consist of a context variable or are built by concatenating to an LF context an LF variable declaration.

We can declare a context schema by $\mathbf{some}\ [\mathtt{y}_1{:}\mathcal{A}'_1,\ \ldots,\ \mathtt{y}_k{:}\mathcal{A}'_k]\ \mathbf{block}\ \mathtt{x}_1{:}\mathcal{A}_1,\ \ldots,\ \mathtt{x}_n{:}\mathcal{A}_n$. An LF context that satisfies this schema if every LF variable declaration is an instance of $\mathbf{block}\ \mathtt{x}_1{:}\mathcal{A}_1,\ \ldots,\ \mathtt{x}_n{:}\mathcal{A}_n$ where we have chosen a suitable instantiation for all the variables $\mathtt{y}_1{:}\mathcal{A}'_1,\ \ldots,\ \mathtt{y}_k{:}\mathcal{A}'_k$.

Substitutions in closures are either empty, written as ^, or a weakening substitution $\ldots$, which we use in practice to transition from a context to a possible extension. Substitutions can also be built by extending a substitution σ with a LF term $\mathcal{M}$; in fact, in general substitutions may be extended with an n-ary tuple of LF terms that provide an instantiation for LF declarations of type $\mathbf{block}\ \mathtt{x}_1{:}\mathcal{A}_1,\ \ldots,\ \mathtt{x}_n{:}\mathcal{A}_n$.

Contextual types and terms pair a LF context together with an LF type or LF term, respectively. As LF contexts are first-class in Beluga, they form valid contextual objects. LF contexts are in general classified by a schema that allows us to state an invariant the LF context satisfies. We refer to the schema by name.

Finally, we come to Beluga's computation language. It allows us to make statements about contextual types and objects and we highlight them in the colour blue. It is in many ways similar to standard functional programming languages with two exceptions:

first, contextual types and objects are the special domain and we support also pattern matching. We only describe here the part of Beluga's term language that is relevant for our example. Beluga programs are constructed using two forms of abstractions: $\Lambda\ \vec{X} \Rightarrow \mathcal{E}$ (abstraction over contextual variables) and **fn** $\vec{x} \Rightarrow \mathcal{E}$ (abstraction over program variables). The remaining constructs are mostly standard in functional programming. We support function application and case statements with pattern matching. In general, branches may be empty, if we try to analyse a Beluga term that has the empty type. This corresponds to an impossible case. We also support let-expressions in our surface language that are translated internally into case expressions with one branch. Beluga supports pattern matching on both inductively defined objects as well as contextual LF objects. Last, we can embed contextual objects into Beluga programs.

In addition to indexed dependent function space, `{X:`$\mathcal{U}$`}`$\mathcal{T}$, Beluga's type language supports simple types, embedding contextual types and (co)inductive definitions, described as `a` $\mathcal{C}_1 \ldots \mathcal{C}_n$. Note that in `{X:`$\mathcal{U}$`}`$\mathcal{T}$, we make `X` explicit and hence any computation-level expression of that type expects first an object of type $\mathcal{U}$. We also allow a limited form of implicit type annotation for context variables; by writing `(X:c)`$\mathcal{T}$, where `c` denotes a context schema, we can abstract over the context variable `X` and declare its context schema, while keeping `X` implicit.