Making Logical Relations More Relatable (Proof Pearl)

Emmanuel Suárez Acevedo emsu@seas.upenn.edu University of Pennsylvania USA Stephanie Weirich sweirich@seas.upenn.edu University of Pennsylvania USA

Abstract

Mechanical proofs by logical relations often involve tedious reasoning about substitution. In this paper, we show that this is not necessarily the case, by developing, in Agda, a proof that all simply typed lambda calculus expressions evaluate to values. A formalization of the proof is remarkably short (~40 lines of code), making for an excellent introduction to the technique of proofs by logical relations not only on paper but also in a mechanized setting. We then show that this process extends to more sophisticated reasoning by also proving the totality of normalization by evaluation. Although these proofs are not new, we believe presenting them will empower both new and experienced programming language theorists in their use of logical relations.

Keywords: logical relations, totality, Agda, normalization by evaluation

1 Introduction

Logical relations are a versatile tool for proving complex properties about programs and programming languages, making them an integral part of programming languages research. There are many examples of proofs by logical relations [13, 23, 24, 28, etc.], one of these being a proof of *normalization* for the simply typed lambda calculus (STLC): if a term is well-typed, then it can be reduced (i.e. computed) to a "normal form", a term that cannot be reduced any further.

Typically, we model the semantics of lambda calculi using a set of rules for reducing terms, featuring a key rule known as the β -reduction:

$$(\lambda x.t)s \rightarrow t[s/x]$$

This rule describes how the application of an abstraction to a term $(\lambda x.t)s$ reduces to the body of the abstraction t with s substituted for every instance of the variable x in t (an operation we write as t[s/x]).

Normalization, according to these reduction rules, is a well-known result for the simply-typed lambda calculus [28]. This proof cannot be shown by simple induction on derivations — instead it requires the use of a proof technique known as a *logical relation*. One must define, by recursion over the structure of types, a predicate that holds for all

normalizing terms. Then one shows the fundamental theorem, which states that this predicate holds for all well-typed terms

Many tutorials on logical relations start by using this technique to show normalization for *weak head reduction* [14, 21, 26, etc.]. Weak head reduction is a limited form of reduction that does not reduce inside the bodies of abstractions. It corresponds to an evaluation semantics for the simply typed lambda calculus. This restriction simplifies the overall normalization argument making it an appropriate introduction to the general technique.

However, although there is a growing corpus of programming languages textbooks that are based on mechanized proofs [22, 29] and work in conjunction with a proof assistant to make sure that students understand each step of the argument, there are not enough mechanized tutorials of proofs based on logical relations aimed at beginning researchers.

We believe that the issue is that in a mechanized setting, the structure of logical relations proofs can be obscured by the many substitution lemmas that are required. While an informal introduction can elide these details, they must be included in a mechanized development. Indeed, Abel et al. [2] note that substitution lemmas are often the most cumbersome part of formalizing a proof by logical relations, proposing the technique as a benchmark for mechanizing metatheory. They quote Altenkirch [5], who remarks:

"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."

But, it doesn't have to be this way. In this paper, we observe that we can introduce the technique of logical relations without needing a single substitution lemma. Furthermore, our approach is based on the standard introductory example: showing that evaluation of the simply typed lambda calculus is total.

Our key insight is the use of an *environment-based natural semantics* [16], also called a big-step semantics, to describe

1

the evaluation of lambda terms. In this setting, our mechanized totality proof is remarkably concise. In Agda, the proof fits entirely in a single column of this paper (Figure 3). The overall structure of the semantics so directly coincides with the structure of the proof that the fundamental lemma for the logical relation needs no auxiliary lemmas (substitution or otherwise). We present this proof in §2, which can be read as a tutorial introduction to logical relations.

Our proof is not exactly new. In essence, it is a simplification of a much longer proof related to the *normalization by evaluation* (NbE) algorithm [7]. Coquand and Dybjer [11] observe that this algorithm can be used for a proof of full normalization for STLC—normal forms can be computed through evaluation. Our simple proof can be viewed as a restriction of this argument to weak head reduction.

What this means is that we can *also* use a logical relations argument to show **full normalization for STLC** without defining or reasoning about substitution. In §3 we extend the tutorial proof to include this result. Our proof is based on an informal presentation given in Abel [1], but we find that in Agda, this proof is as nearly as approachable in a mechanized setting as it would be on paper. This section demonstrates that this technique is not limited to simple properties that only apply to closed terms.

Overall, these two examples demonstrate that we can prove complex properties using a logical relations argument without needing substitutions or substitution lemmas, a result that we hope empowers researchers in both mechanizing future proofs and in understanding the proof technique. As far as we can tell, this is the first presentation with the explicit goal of introducing the technique of logical relations in a mechanized setting.

As these results are especially relevant in a mechanized setting, we have written this paper as a literate Agda script, presenting both proofs in Agda. Every highlighted line of code rendered here has been checked by Agda, and we have only omitted imports, precedence declarations for mixfix syntax, and repeated code. We assume the reader has basic familiarity with Agda, but the content itself is self-contained and we explain more advanced Agda features as they are encountered.

We have included the content of the appendix in our supplemental material for the reviewers, these proofs make up all of the literate Agda code presented in the main body of the paper.

2 Totality of evaluation of STLC

A property that we are generally interested in proving about a program is that if it can be assigned a type, then its execution is well-defined. Equivalently, we can prove that its evaluation is total. Proving that evaluation is total is related

```
data Type : Set where
   bool: Type
   \implies_: Type \rightarrow Type \rightarrow Type
variable S T : Type
data Ctx: Set where
   \emptyset: Ctx
   : Ctx \rightarrow Type \rightarrow Ctx
variable \Gamma: Ctx
- Intrinsically-scoped de Brujin indices
data \ni : Ctx \rightarrow Type \rightarrow Set where
   zero: \Gamma \cdot: T \ni T
   suc : \Gamma \ni T \longrightarrow \Gamma :: S \ni T
variable x : \Gamma \ni T
- Intrinsically-typed terms
data \vdash: Ctx \rightarrow Type \rightarrow Set where
   true false : \Gamma \vdash bool
   var : \Gamma \ni T \rightarrow \Gamma \vdash T
   \lambda : \Gamma : S \vdash T \rightarrow \Gamma \vdash S \Rightarrow T
   \underline{\,\,\,}:\Gamma \vdash \mathsf{S} \Longrightarrow \mathsf{T} \to \varGamma \vdash \mathsf{S} \to \varGamma \vdash \mathsf{T}
   if then else : \Gamma \vdash \mathsf{bool} \to \Gamma \vdash \mathsf{T} \to \Gamma \vdash \mathsf{T} \to \Gamma \vdash \mathsf{T}
variable r s t : \Gamma \vdash T
```

Figure 1. STLC

to proving normalization with weak head reduction, though the two properties are distinct and should not be confused. In proving that evaluation is total, we do not concern ourselves with the normal form that any well-typed term can be reduced to (i.e. computed), but rather only on showing that the evaluation of the term (i.e. the computation itself) is well-defined.

We prove that evaluation is total for STLC. To do so, we must first represent STLC in Agda and model the behavior (the semantics) of an STLC term.

2.1 Embedding of STLC in Agda

We prove that evaluation of STLC is total for a minimal subset of the language for brevity, shown in Figure 1. The proof has no significant complexity added when adding more familiar constructs such as sums or products. We use booleans (bool) for our base type and a function type ($S \Rightarrow T$). We consider variables (var x), abstractions (\hbar t), and application ($r \cdot s$). Additionally, we have the boolean constants true and false, along with conditional branching (if r then s else t).

We represent variable bindings using de Brujin indices [12], and a typing context serves as a list of types with each index into the list representing a variable. We define typing contexts as either the empty context \emptyset or an extension to

¹In contrast, the proof in Pierce et al. [22] requires nineteen lemmas to show an equivalent result.

a context Γ : T. We use a lookup judgement into a context Γ \ni T for de Brujin indices so that they may be intrinsically scoped, using the constructors zero and suc suggestively.

A term t is an intrinsically typed Agda expression; we do not consider raw terms that may be ill-typed and represent terms by their typing derivation. We choose an intrinsically typed representation to simplify our discussion in §4. The decision is irrelevant for the proof that evaluation is total, however. We can prove that evaluation is total with an extrinsic representation of terms with few changes to the formalization we present here. Surprisingly, this remains the case even if using raw bindings (e.g. strings) to represent variables.² Consequently, the proof we present here could be introduced as soon as the simply typed lambda calculus is first taught in material using proof assistants for education, where it is common to first use an embedding of the STLC using raw bindings for variables to improve readability (e.g. as done by Wadler et al. [29] and Pierce et al. [22]).

To make some of our relations and theorems more succinct, we make use of a feature of Agda that allows us establish metavariables. For example, variable $x : \Gamma \ni T$ indicates the metavariable x will be used for the de Brujin indices making up variable bindings in our representation of the STLC. The first such relation that takes advantage of this feature is our evaluation relation, which we define in the next section.

2.2 Natural semantics

We can model the behavior of a program with a small-step relation (each "step" a program takes in its execution) or a big-step relation (the execution of a program as a whole).

We often describe the semantics of STLC through the use of substitution, an operation that replaces every instance of a variable in a term for another term in a β -reduction. A β -reduction closely resembles a single step that a program may take in its execution, so the rule is often used to describe a small-step operational semantics.

While a small-step semantics almost always uses substitution to model function application, a big-step semantics may be defined equivalently using an environment or with substitutions. Big-step semantics are commonly used to reason about programs as noted by Charguéraud [10] when they claim that *big-step is not dead*.

The semantics we use (shown in Figure 2) is called a natural semantics [16], modeling the execution of a term through its evaluation. It is a big-step semantics that uses an environment γ to evaluate a term t to a result value a, notated $\gamma \mid t \parallel a$. As this semantics does not use substitution, we can avoid having to reason about the operation.

We model the evaluation of a term as a relation and not a function because we cannot write partial functions in Agda,

```
mutual
   - Environments
   Env : Ctx \rightarrow Set
   Env \Gamma = \forall \{T\} \rightarrow \Gamma \ni T \rightarrow Domain T
   - Domain of evaluation
   data Domain: Type → Set where
       - Booleans
      true false: Domain bool
      - Closures
       \langle \lambda_{-} \rangle_{-} : \Gamma : S \vdash T \rightarrow Env \Gamma \rightarrow Domain (S \Rightarrow T)
variable \gamma \delta : Env \Gamma
variable a b d : Domain T
- Extending an environment
\_++\_ : Env \Gamma \rightarrow Domain T \rightarrow Env (\Gamma :: T)
(\gamma ++ a) zero = a
(\gamma ++ a) (suc x) = \gamma x
- Evaluation of terms
data \_|\_\downarrow\!\!\!\!\!\bot: Env \Gamma \to \Gamma \vdash T \to Domain T \to Set where
   evalTrue : γ | true ↓ true
   evalFalse : \gamma | false | false
   evalVar : \gamma \mid \text{var } \mathbf{x} \parallel \gamma \mathbf{x}
   evalAbs : \gamma \mid \lambda t \downarrow \langle \lambda t \rangle \gamma
   evalApp:
      \gamma \mid r \downarrow \langle \lambda t \rangle \delta
       \rightarrow \gamma \mid s \downarrow a
       \rightarrow \delta ++ a | t \downarrow b
       \rightarrow \gamma \mid r \cdot s \parallel b
   evallfTrue:
      y | r ↓ true
       \rightarrow \gamma \mid s \downarrow a
       \rightarrow \gamma | if r then s else t \downarrow a
   evalIfFalse:
      \gamma \mid r \downarrow \text{ false}
       \rightarrow \gamma \mid t \parallel b
       \rightarrow \gamma | if r then s else t \downarrow b
```

Figure 2. Natural semantics

and we do not know if evaluation is total – this is the property we want to prove!³ If we were writing these semantics as a function in a language such as Haskell, however, they would be comparable to an interpreter that one might write for the simply typed lambda calculus (and in fact, the denotational semantics we discuss in §4 are just that).

 $^{^2\}mathrm{We}$ include a formalization of the totality of evaluation for STLC using an extrinsic representation with raw bindings in appendix C to make this apparent.

³As we are using an intrinsically-typed representation, Agda can check that such a function is in fact total, though we do not take advantage of this as we want to present a proof independent of our use of intrinsic typing.

Environments are defined mutually with a domain of evaluation that makes up the set of fully evaluated terms. For our minimal STLC, the only elements in this domain are true, false, and closures ($\langle \lambda t \rangle \delta$), i.e. an abstraction paired with a saved environment [17].⁴ Both environments and domain elements are well-typed by construction, an environment γ is typed according to a context Γ and can only map variables that are present in Γ to domain elements that are themselves well-typed. Domain elements are typed independently of contexts, as we close terms with environments.

Definitionally, environments have no real difference from substitutions, with their distinguishing factor being how they are used. Instead of applying a substitution to a term to reduce it, we perform a sort of delayed substitution where we substitute a variable in a term as we evaluate the term itself.

We evaluate terms to a separate domain instead of reducing the term to a value as we might do in a semantics described using substitutions. This is because in evaluating an abstraction (evalAbs), we wish to save the environment that the abstraction is being evaluated under to form a closure. This allows us to use the environment later on to continue evaluating the body of the abstraction. We do so in the case of application (evalApp), where we evaluate the term being applied to a closure and then extend its environment to continue evaluating its body. Note that as we evaluate the term being applied to extend the closure's environment, these are semantics call-by-value.

2.3 Proof by logical relation

We now turn to the property we wish to prove, that the execution of a program is well-defined. With our semantics, this is equivalent to proving that evaluation of STLC is total. We consider the evaluation of closed terms only as would be the case with weak head reduction. We do not evaluate inside the bodies of abstractions, therefore we do not evaluate open terms.

```
empty : Env \varnothing empty () 
 \Downarrow-well-defined : \varnothing \vdash T \rightarrow Set 
 \Downarrow-well-defined t = \exists [a] empty | t \Downarrow a
```

We cannot prove that evaluation is well-defined for a well-typed term by direct induction on the typing derivation because our induction hypotheses would not be strong enough in the case of application $r \cdot s$.

With our induction hypotheses, we would have that the evaluation is well-defined for the term r and that it evaluates to a closure $\langle \lambda, t \rangle \delta$. We would additionally have that evaluation is well-defined for the term s, evaluating to some

```
    Semantic types (logical predicate)

[\![ \_ ]\!]: \forall (T:Type) \rightarrow (Domain T \rightarrow Set)
\llbracket S \Rightarrow T \rrbracket (\langle \lambda t \rangle \delta) =
   \forall \{a\} \rightarrow a \in \llbracket S \rrbracket
   \rightarrow \exists [b] \delta ++ a | t \parallel b \times b \in [T]
- Semantic typing for environments
\models : (\Gamma : \mathsf{Ctx}) \to \mathsf{Env} \ \Gamma \to \mathsf{Set}
\Gamma \models \gamma = \forall \{T\} \rightarrow (x : \Gamma \ni T) \rightarrow \gamma x \in \llbracket T \rrbracket
- Extending semantically typed environments
^{\wedge}: \Gamma \models \gamma \rightarrow a \in \llbracket T \rrbracket \rightarrow \Gamma : T \models \gamma ++ a
(⊧y ^ sa) zero = sa
(\models \gamma \land sa) (suc x) = \models \gamma x
- Semantic typing for terms
semantic-typing : \Gamma \vdash T \rightarrow Set
semantic-typing \{\Gamma\} \{T\} t =
   \forall \{ \gamma : \mathsf{Env} \ \Gamma \} \to \Gamma \models \gamma \to \exists [ \ \mathsf{a} \ ] \ \gamma \mid \mathsf{t} \ \downarrow \mathsf{a} \times \mathsf{a} \in \llbracket \ \mathsf{T} \ \rrbracket
syntax semantic-typing \{\Gamma\} \{T\} t = \Gamma \models t :: T
- Syntactic typing implies semantic typing
fundamental-lemma : \forall (t : \Gamma \vdash T) \rightarrow \Gamma \models t :: T
fundamental-lemma true _ = true, evalTrue, tt
fundamental-lemma false _ = false , evalFalse , tt
fundamental-lemma (var x) \{y\} \models y =
   y x , evalVar , ⊧y x
fundamental-lemma (\lambda t) \{ \gamma \} \neq \gamma =
   \langle \lambda t \rangle \gamma, evalAbs, \lambda sa \rightarrow fundamental-lemma t (\models \gamma \land sa)
fundamental-lemma (r \cdot s) \models \gamma
   with fundamental-lemma r \models \gamma
... \mid \langle \lambda t \rangle \delta, r \parallel, sf =
   let (a, s \downarrow , sa) = fundamental-lemma s \models y in
   let (b , eval-closure , sb) = sf sa in
   b, evalApp r ↓ s ↓ eval-closure, sb
fundamental-lemma (if r then s else t) \models y
   with fundamental-lemma r ⊧y
... | true , r↓, _ =
   let (a, s \downarrow , sa) = fundamental-lemma s \models y in
   a, evallfTrue r↓ s↓, sa
... | false , r↓, _ =
   let (b, t \downarrow , sb) = fundamental-lemma t \models y in
   b, evallfFalse rll tll, sb
- Evaluation is total
\downarrow \text{-total} : \forall (t : \varnothing \vdash T) \rightarrow \downarrow \text{-well-defined } t
∥-total t =
   let (a, t \parallel a, _) = fundamental-lemma t (\lambda ()) in
   a,t∥a
```

Figure 3. Full proof that evaluation of STLC is total

⁴We take advantage of Agda's support for overloading inductive constructors here, so that we can reuse false and true instead of using new names.

domain element a. However, we want to show that the evaluation of the closure's body t is itself well-defined when the environment δ is extended with a, i.e. that there exists some b such that δ ++ a | t \parallel b. This is something that is not given to us by our induction hypothesis.

We turn instead to the use of a logical predicate on the domain of evaluation (i.e. Domain \rightarrow Set). For any logical predicate A, we use Agda's set membership notation $a \in A$ to refer to a domain element a that satisfies A.

A logical predicate is a unary logical relation. A logical relation is always defined recursively on types. At each type, the relation describes the "logical" interpretation of that type. In other words, we describe the behavior that we would *expect* a program to have at that type. In particular, at function types we generally expect related arguments to give us related results: if a function is given an argument that satisfies a logical relation, the result should also satisfy the relation.

We follow this formula to define the logical predicate $[\![T]\!]$, which serves as a "semantic type" describing the well-defined evaluation of terms.

At our boolean type, we have that evaluation is well defined for all domain elements (true and false), so we use Agda's unit type \top .

At our function type, we define the logical predicate on closures such that a closure $\langle \bar{\mathcal{X}} \ t \ \rangle \ \delta$ has a semantic function type $[\![\ S \Rightarrow T \]\!]$ if evaluation is well-defined for the body t of the closure when its environment δ is extended with any arbitrary $a \in [\![\ S \]\!]$ such that the evaluated body satisfies the logical predicate described by $[\![\ T \]\!]$ (i.e. δ ++ a | t $\|$ b and b $\in [\![\ T \]\!]$).

This is one the key components mentioned earlier of a proof by logical relations; where a semantic function type takes related arguments ($a \in [S]$) to related results ($b \in [T]$). This structure of our logical relation solves the issue we described in the case of application.

We use this logical predicate to prove a stronger property than just proving that the evaluation of a term is well-defined, instead we prove that a well-typed term satisfies the the semantic typing judgement $\Gamma \models t :: T$. Given an environment that is semantically typed according to the context Γ , we expect the evaluation of the term t in that environment to be well-defined and for its evaluation a to be semantically typed, i.e. $a \in A$. Proving this property gives us a strong enough induction hypothesis to prove that evaluation is total.

We define a semantically typed environment as an environment that is made up of domain elements that are themselves semantically typed, notated $\Gamma \models \gamma$.

Having set up a logical predicate and a semantic typing judgement for terms, we can prove the fundamental lemma of logical relations: if a term is well-typed, it is also semantically typed. We prove this lemma by induction on our typing derivation. The boolean constants true and false trivially satisfy the logical predicate described by the semantic boolean type. In the case of a variable, we use the fact that the environment is semantically typed. In the case of application, we use the semantic typing of the evaluated subterms to have an appropriately strong induction hypothesis. For conditional branching, our induction hypothesis gives us that either branch is semantically typed regardless of whether the conditional evaluates to true or false. In our abstraction case, we show that the closure an abstraction evaluates to is semantically typed by using our induction hypothesis on an extended semantically typed environment.

This is the familiar structure for a proof of the fundamental lemma of logical relations. It is our abstraction case that has the most notable difference; our induction hypothesis no longer has a need for a substitution lemma.

Putting all of these pieces together, we can now prove that evaluation of closed terms is total in the empty environment as a corollary of the fundamental lemma, given that an empty environment is vacuously semantically typed according to the empty context. This is our entire proof by logical relations. We present it all in Agda (Figure 3), without needing to veer too far from how it might be presented on paper. Now that we have proven that evaluation of STLC is total, we move on to extending the proof.

3 Extending the proof: full normalization of STLC

We extend our proof that evaluation is total to prove normalization of the simply typed lambda calculus; any well-typed term t reduces to a normal form v. We can prove normalization with either weak head reduction or full reduction (reducing under the body of abstractions). In the previous section, proving that evaluation of the simply typed lambda calculus is total is very close to proving normalization with weak head reduction. All that we are missing is determining the normal form of a term from its evaluation.

Determining the normal form of a term from its evaluation is a well-known technique called normalization by evaluation (NbE). Typically it is described as an algorithm, though we can also describe it as a proof of normalization. We prove that evaluation of a well-typed term is well-defined, after which we prove that we can read back a normal form from its evaluation.⁵

Extending our proof that evaluation is total to prove normalization with weak head reduction is not too much additional work: our domain of evaluation is made up of our normal forms. If a term evaluates to a boolean, then its normal form is that boolean. If a term evaluates to a closure, all we need to do is "close" the term by performing the delayed substitution represented by its saved environment.

⁵Though this does not prove the correctness of the technique, something we discuss in §4.

In appendix A, we extend our proof that evaluation of the simply typed lambda calculus is total to prove normalization with weak head reduction. As we have to perform a delayed substitution to determine a normal form, reasoning about normalization involves reasoning about substitutions and related lemmas. For example, we also show that our proof normalization with weak head reduction is sound, for which we need to show that soundness is closed under the substitution operation.

If we extend our proof that evaluation is total to prove normalization with full reductions, the proof resembles the typical algorithm for NbE (which is described with full reductions). The algorithm for NbE interleaves the evaluation of terms and reading back normal forms. As a result, it does not need a substitution operation. This remains the case for a proof by evaluation of normalization with full reductions.

We extend our proof that evaluation is total and prove normalization with full reductions (formalizing the proof presented in Abel [1]). The proof illustrates how using the evaluation of programs to prove properties by logical relations without needing substitutions is not a one-off trick. Indeed, there are sophisticated logical relations arguments that do not require reasoning about substitutions. Consequently, the technique is a valuable way to introduce proofs by logical relations.

The algorithm for NbE can be summarized as follows: first, we evaluate a term in an environment of the variables in its context, then we read back a normal form from its evaluation in this environment.

Converting this to a proof of normalization consists of 1) proving that evaluation in an environment of variables is total for well-typed terms, and 2) proving that we can determine a normal form from this evaluation.

The first step remains mostly the same as in the previous section. We omit our proof that evaluation is total, including only the differences from the previous section.⁶ The most notable difference from the previous section is that we extend our domain of evaluation to include blocked evaluations such as variables, so that we can evaluate a term in an environment of variables.

The second step makes up most of the work in this section. We must extend our semantics with a relation between the evaluation of a term and the normal form we read back. Additionally, we have to modify our semantic types so that we can prove that semantically typed terms can have a normal form read back from their evaluation.

3.1 Switching to an extrinsic representation

However, before proceeding to prove normalization, we must first switch to an extrinsic representation of the simply typed lambda calculus.

```
data Type: Set where
   base: Type
   \implies_: Type \rightarrow Type \rightarrow Type
variable S T : Type
data Ctx: Set where
   \emptyset: Ctx
   : Ctx \rightarrow Type \rightarrow Ctx
variable \Gamma: Ctx
- Raw terms
data Term: Set where
   var: (x: \mathbb{N}) \to Term
   \lambda_: Term \rightarrow Term
   \_\cdot\_:\mathsf{Term}\to\mathsf{Term}\to\mathsf{Term}
variable x: N
variable r s t u v : Term

    Variable lookup

data :: \in : \mathbb{N} \to \mathsf{Type} \to \mathsf{Ctx} \to \mathsf{Set} where
   here : zero :: T \in \Gamma ·: T
   there : x :: T \in \Gamma \rightarrow suc \ x :: T \in \Gamma :: S
- Typing judgement
data \vdash :: : Ctx \rightarrow Term \rightarrow Type \rightarrow Set where
  \vdashvar : x :: T \in \Gamma \rightarrow \Gamma \vdash var x :: T
  \vdash abs : \Gamma :: S \vdash t :: T \rightarrow \Gamma \vdash \lambda t :: S \Rightarrow T
  \vdashapp: \Gamma \vdash r :: S \Rightarrow T \rightarrow \Gamma \vdash s :: S \rightarrow \Gamma \vdash r \cdot s :: T
```

Figure 4. Extrinsic representation of STLC

An intrinsic representation does not complicate our proof that evaluation is total, even if we extend it to prove normalization with weak head reduction. In contrast, an intrinsic representation complicates the proof significantly if we extend it to prove normalization with full reductions. There are a few reasons for this, with the most significant of these being that we will now be including variables in our domain of evaluation (we now evaluate a term in an environment of variables in the process of normalization). With an intrinsically typed representation, we need to carry around a context in our domain of evaluation to represent variables, which results in a need to reason about context extension and renaming.

For simplicity, we switch to an extrinsic representation of terms (shown in Figure 4) in this section. We drop booleans and conditional branching for brevity; adding them back does not affect the overall complexity of our proof. We only consider variables, abstractions, and application, changing our boolean base type for an empty base type base that is inhabited only by variables. Even with these changes, a fair

 $^{^6\}mathrm{We}$ include the full proof, including any omissions in the main body of the paper, in appendix B.

```
mutual
  - Environments
  Env = \mathbb{N} \rightarrow Domain
  - Domain
  data Domain: Set where
     - Closures
     \langle \lambda \rangle : \text{Term} \to \text{Env} \to \text{Domain}
     - Neutral domain elements
     : \mathsf{Domain}^{ne} \to \mathsf{Domain}
  - Neutral domain
  data Domain<sup>ne</sup>: Set where
     - Variables (de Brujin levels)
     lvl: (k: \mathbb{N}) \rightarrow Domain^{ne}
     - Application
     \cdot: \mathsf{Domain}^{ne} \to \mathsf{Domain} \to \mathsf{Domain}^{ne}
variable a b d f : Domain
variable e : Domain^{ne}
variable k : \mathbb{N} - metavariable for de Brujin level
variable \gamma \delta: Env
++ : Env \rightarrow Domain \rightarrow Env
(\underline{\phantom{a}} ++ a) zero = a
(\gamma ++ \_) (suc m) = \gamma m
```

Figure 5. Changes to environments and domains

portion of our proof remains relatively unchanged from the previous section.

Using this representation, we will prove normalization by proving that any term that is well-typed has a normal form. We do so by determining the normal term that a term reduces to by evaluation. We define normal terms mutually with neutral terms:

mutual

```
data Normal : Term → Set where normalAbs : Normal v \to Normal \ (\hbar v) neutral : Neutral u \to Normal \ u

data Neutral : Term → Set where neutralVar : Neutral (var x) neutralApp : Neutral u \to Normal \ v \to Neutral \ (u \cdot v)
```

Normal terms are terms in their normal form. They are either abstractions whose bodies are normal terms, or neutral terms. Neutral terms are either variables or the application of a neutral term to a normal term.

Figure 6. Changes to natural semantics

3.2 Natural semantics revisited

In the previous section, our natural semantics did not evaluate the bodies of abstractions. As we are now proving normalization with full reductions, this is no longer enough. However, we do not change our natural semantics to evaluate inside the bodies of abstractions. Instead, we change our domain of evaluation in Figure 5.

Our domain of evaluation is no longer terms that are fully evaluated, but rather terms that are in the *process* of being evaluated to have a normal form read back. Later, we will define a relation for reading back a normal form from the evaluation of a term, but we focus on the changes to our domain of evaluation first.

We extend our domain to include "blocked" evaluations, which we call the neutral domain (Domain^{ne}). A domain element 'e that cannot be evaluated any further is either a variable (Ivl k) or the application of an element in the neutral domain to any element in the domain (e · d). In our domain of evaluation, we use de Brujin levels instead of de Brujin indices to represent variables. This is because de Brujin levels act more as constants, allowing us to avoid having to rename variables as we evaluate terms.

With these changes to our domain of evaluation, our natural semantics in Figure 6 remain almost the same as before. The main difference is that we now generalize evalApp with a condition for well-defined application. This is because there are now two cases for well-defined application, with appClosure being the case we had before. Our other case, appNeutral, is for the application of a domain element to a neutral domain element. In this case, the evaluation simply remains blocked.

We now turn to the question of evaluating inside the bodies of abstractions. Without this, our semantics seem almost incomplete, as we are still evaluating abstractions to closures. We now describe how to make the semantics feel

```
variable n: \mathbb{N} - Scope
- Converting a de Brujin level to a de Brujin index
|v| \rightarrow idx : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
lvl \rightarrow idx k n = n - suc k
mutual
  - Reading back a normal term
  data |\uparrow\rangle: \mathbb{N} \to Domain \to Term \to Set where
     ↑closure:
        \delta ++ 'lvln|t\parallela
        \rightarrow n | a \uparrow v
        \rightarrow n | \langle \lambda t \rangle \delta \uparrow \lambda v
     - Reading back a neutral term
  data | \uparrow \uparrow^{ne} : \mathbb{N} \to \mathsf{Domain}^{ne} \to \mathsf{Term} \to \mathsf{Set} where
     | lv| : n | lv| k | l^{ne} var (| lv| \rightarrow idx k n)
     ↑app:
        n | e ↑ ne u
        \rightarrow n | d \uparrow v
        \rightarrow n | e · d \uparrow^{ne} u · v
```

Figure 7. Reading back a normal form from the evaluation of a term

more "complete" by introducing the notion of a reading back a normal form from an evaluated term.

3.3 Reading a normal form back from an evaluated term

We now describe the conditions for reading back a normal term v from the evaluation d of a term t, notated $n \mid d \uparrow v$. It is defined mutually with the reading back of a neutral term, notated $n \mid e \uparrow \uparrow^{ne} u$.

Both relations need to know the overall "scope" of the original term t. This is the total number n of variables in the term's context. We use this value to convert a de Brujin level to its corresponding de Brujin index in | v | level.

In rule \uparrow closure, we actually evaluate the body of the closure to read back a normal term. This was not possible with weak head reduction, as our domain of evaluation did not include variables. With our new domain, we can evaluate the body of the closure by extending its environment with a "fresh" variable (the first de Brujin level not in scope). After we have evaluated the body of the closure, we can read back a normal term. This rule is why we switched to de Brujin levels in our domain of evaluation: we can evaluate the body of the closure without needing to introduce a renaming operation.

With these changes, we can now turn to the property that we want to prove, normalization. For any term, we wish to show that it can be evaluated in an environment consisting of the variables in its scope (all converted to their corresponding de Brujin levels) and then have a normal form read back.

```
\begin{split} & \text{id} x \!\!\to\!\! \text{IvI} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ & \text{id} x \!\!\to\!\! \text{IvI} \text{ i } n = n \text{ - suc i} \\ & \text{env} : \mathbb{N} \to \text{Env} \\ & \text{env n i = 'IvI (id} x \!\!\to\!\! \text{IvI i n)} \\ & \text{\_has-normal-form} <\_>\_: \text{Term} \to \mathbb{N} \to \text{Term} \to \text{Set} \\ & \text{t has-normal-form} < n > v = \\ & \exists \text{[ a ] env n | t} & \exists x \text{ n | a} & \forall v \end{split}
```

Note that any term that is read back from the domain of evaluation is a normal term, something that we can prove quickly by induction on the read back relation. (We include this proof at the end of appendix B). For this reason, we only need to prove that we can read back a term in the first place according to the relation.

3.4 Asking more of semantic types with candidate spaces

Proving that the evaluation of a well-typed term is total is the same as before. The only difference is that we are now evaluating the term in an environment of variables instead of the empty environment. This does not change the fundamental lemma at all (and we omit its proof), as it is generalized over any environment that is semantically typed. As a result, all we need to do here is prove that an environment of variables is semantically typed. This will be a consequence of proving that if the evaluation of a term is semantically typed, we can read back a normal form from its evaluation. We begin to address this matter in this section.

To prove that we can read back a normal form from the evaluation of a term if it is semantically typed, we must define a "candidate space" (Figure 8) of elements in the domain that can actually have a normal form read back from them. We define a "bottom" of the candidate space $\mathbb B$ (the subset of the neutral domain that can have a neutral term read back) and a "top" of the candidate space $\mathbb T$ (the subset of the domain that can have a normal term read back). The bottom of the candidate space is a subset of the top of the candidate space (as elements in the neutral domain are a subset of the domain and neutral terms are a subset of normal terms).

For convenience, we define LogPred to be Domain \rightarrow Set, as we will be using logical predicates more in this section. In particular, the top and bottom of the candidate space are themselves logical predicates.

We now update our semantic types so that they inhabit this candidate space, this is referred to as "asking more" of semantic types [13].

```
LogPred = Domain \rightarrow Set
- Bottom of candidate space
- (neutral term can be read back)
\mathbb{B}: LogPred
\mathbb{B} ('e) = \forall n \rightarrow \exists[u] n | e \uparrow<sup>ne</sup> u
\mathbb{B} = \bot
- Top of candidate space
- (normal term can be read back)
\mathbb{T}: LogPred
\mathbb{T} d = \forall n \rightarrow \exists [v] n \mid d \uparrow v
- Bottom of space is a subset of top
\mathbb{B} {\subseteq} \mathbb{T} : d \in \mathbb{B} \to d \in \mathbb{T}
\mathbb{B} \subseteq \mathbb{T} \{ (e) \}  eb n
  with eb n
... | u , eîu =
  u, ↑neutral e↑u
```

Figure 8. Candidate space for reading back a normal term

```
- Semantic function space 

_→_: LogPred → LogPred → LogPred 

(A → B) f = \forall \{a\} \rightarrow a \in A \rightarrow \exists [b] f \cdot a \Downarrow b \times b \in B 

- Semantic types (logical relation) 

[_]: Type → LogPred 

[ base ] = B 

[ S ⇒ T ] = [ S ] → [ T ]
```

Figure 9. Changes to semantic types

3.5 Changes to semantic types

The only domain elements that may inhabit our semantic base type are neutral domain elements (e.g. variables), we expect closures to inhabit the semantic function type. If a neutral domain element is semantically typed, we wish to be able to read back a neutral term from it. Therefore, we insert the bottom of the candidate space $\mathbb B$ at our base semantic type.

As a result of this change to our logical predicate, semantic types are now restricted so that they inhabit the candidate space, i.e. for any type T, we have that $\mathbb{B} \subseteq \llbracket T \rrbracket \subseteq \mathbb{T}$). This is not obvious, and we have to prove it explicitly in the next section.

In restricting our semantic types so they inhabit the candidate space, proving that the evaluation of a term is semantically typed immediately yields that we can read back a normal term.

Our semantic function type remains unchanged, though we generalize it to use a semantic function space $A \longrightarrow B$.

```
\begin{split} \mathbb{B} \subseteq \mathbb{T} \longrightarrow \mathbb{B} : \text{`} e \in \mathbb{B} \longrightarrow \text{`} e \in \mathbb{T} \longrightarrow \mathbb{B} \\ \mathbb{B} \subseteq \mathbb{T} \longrightarrow \mathbb{B} \{e\} \text{ eb } \{d\} \text{ dt } = \\ \text{`} (e \cdot d), \\ \text{appNeutral }, \\ \lambda \text{ n} \longrightarrow \\ \text{let } (u \text{ , e} \cap u) = \text{eb n in} \\ \text{let } (v \text{ , d} \cap v) = \text{dt n in} \\ u \cdot v \text{ , } \cap \text{app e} \cap u \text{ d} \cap v \end{split}
```

Figure 10. First property of candidate space

We do so because we will be reusing the semantic function space in our reasoning in the next section when proving that semantic types inhabit the candidate space. The semantic function space is equivalent to our former presentation, however, and we could have defined our semantic function type this way if we had wanted to.

Note that here, we are now using $f \cdot a \Downarrow b$ in our semantic function type instead of $\delta ++ a \mid t \Downarrow b$ as before. This is because we have extended our semantics to include more cases for well-defined application in the domain.

3.6 Semantically typed domain elements inhabit the candidate space

The top and bottom of the candidate space are chosen such that a few key properties hold, by using these properties we are able to prove that we can read back a normal form from an evaluated term if it is semantically typed. The first property (Figure 10) is that the bottom of the candidate space $\mathbb B$ is a subset of the semantic function space $\mathbb T \longrightarrow \mathbb B$. Intuitively, this property holds because for any neutral domain element $e \in \mathbb B$ we can apply it to any domain element $e \in \mathbb B$ we can apply it to any domain element $e \in \mathbb B$ and have a neutral domain element that can still have a neutral term read back (the application of each domain element's corresponding term that is read back).

For the second property, we first prove a separate lemma that all variables are in **B**. This is because any de Brujin level can be read back to its corresponding de Brujin index.

```
|v| \in \mathbb{B} : \forall k \to `lv| k \in \mathbb{B}
|v| \in \mathbb{B} k n = var(|v| \to idx k n), \uparrow |v|
```

The second property (Figure 11) is that any object in the semantic function space $\mathbb{B} \longrightarrow \mathbb{T}$ is a subset of the top of the candidate space \mathbb{T} . For any domain element $d \in \mathbb{B} \longrightarrow \mathbb{T}$, we have by the properties of the semantic function space that the evaluation of its application to the first variable not in scope (which is in \mathbb{B}) can have a normal form read back. We do a case analysis on the evaluation of this application to determine the normal form of the domain element d.

Using these properties, we are now able to prove that semantically typed domain elements inhabit the candidate space; for any type T, we have that T T is a subset of T.

```
\begin{split} \mathbb{B} & \longrightarrow \mathbb{T} \subseteq \mathbb{T} : d \in \mathbb{B} \longrightarrow \mathbb{T} \longrightarrow d \in \mathbb{T} \\ \mathbb{B} & \longrightarrow \mathbb{T} \subseteq \mathbb{T} \left\{ \left\langle \mathring{\mathcal{X}} \ t \ \right\rangle \ \gamma \right\} \ pf \ n \\ & \text{with pf (Ivl} \in \mathbb{B} \ n) \\ & ... \ | \ d \ , \ app Closure \ eval-closure \ , dt \\ & \text{with dt n} \\ & ... \ | \ v \ , \ d \| v = \\ & \mathring{\mathcal{X}} \ v \ , \ \| \ closure \ eval-closure \ d \| v \\ & \mathbb{B} \longrightarrow \mathbb{T} \subseteq \mathbb{T} \ \left\{ \ ' \ e \right\} \ pf \ n \\ & \text{with pf (Ivl} \in \mathbb{B} \ n) \\ & ... \ | \ _ \ , \ app Neutral \ , et \\ & \text{with et n} \\ & ... \ | \ u \cdot v \ , \ \| \ neutral \ ( \| \ app \ e \| \ u \ ) = \\ & u \ , \ \| \ neutral \ e \| \ u \end{aligned}
```

Figure 11. Second property of candidate space

The result follows in Figure 12 from the properties we have shown so far along with two lemmas we prove mutually. The first is that $\mathbb{T} \longrightarrow \mathbb{B}$ is a subset of $[\![S \Rightarrow T]\!]$, which we use to prove that $[\![S \Rightarrow T]\!]$ is a subset of any semantic type. The second is that $[\![S \Rightarrow T]\!]$ is a subset of $[\![B \longrightarrow T]\!]$, which we use to prove that any semantic type is a subset of $[\![T]\!]$.

3.7 Establishing normalization

We prove normalization in Figure 13. We use the fundamental lemma to prove that the evaluation of a well-typed term in an environment of variables is well-defined.

We prove that the environment of variables is semantically typed by using the fact that all elements of $\mathbb B$ are semantically typed, composed with the fact all variables are elements of $\mathbb B$.

By the fundamental lemma, we also have that the evaluation of a well-typed term is semantically typed. As we have just shown, this implies that we can read back a normal form from the evaluation of the term.

We therefore prove normalization of STLC, as for any term that is well-typed its evaluation is well-defined and we can read back a normal form from its evaluation.

4 Correctness

Our formalization has almost a direct correspondence to how it may be presented in an informal mathematical presentation. For our proof that evaluation is total, the main difference is that we use an intrinsic representation. The formalization becomes more complicated when extended to prove normalization with full reduction, though remains nearly as approachable as it would if presented informally. This brings us to the question: is it okay to avoid substitutions as we have done here?

Natural semantics are a well established model for the semantics of programming languages, so it makes sense to use a natural semantics to prove that evaluation of the simply

```
mutual
     \mathbb{T} {\longrightarrow} \mathbb{B} {\subseteq} \llbracket \_ {\Rightarrow} \_ \rrbracket : \forall \ S \ T \to f \in \mathbb{T} \longrightarrow \mathbb{B} \to f \in \llbracket \ S \Rightarrow T \ \rrbracket
    \mathbb{T} \longrightarrow \mathbb{B} \subseteq \llbracket S \Rightarrow T \rrbracket \text{ pf sa}
          with [S]⊆T sa
     ... | at
         with pf at
    ... | 'e, app-eval, eb
         with B⊆ Teb
    ... | se =
          'e,app-eval,se
     \llbracket \Rightarrow \rrbracket \subseteq \mathbb{B} \longrightarrow \mathbb{T} : \forall \ S \ T \rightarrow f \in \llbracket \ S \Rightarrow T \ \rrbracket \rightarrow f \in \mathbb{B} \longrightarrow \mathbb{T}
     \llbracket S \Rightarrow T \rrbracket \subseteq \mathbb{B} \longrightarrow \mathbb{T} \text{ sf } \{\text{`e}\} \text{ eb}
          with sf (\mathbb{B}\subseteq [S] eb)
     ... | d , app-eval , sd =
          d, app-eval, [T]⊆T sd
     - Any object that can have a neutral form read back
    - is semantically typed
    \mathbb{B}\subseteq \llbracket \ \rrbracket : \forall \ \mathsf{T} \to `e \in \mathbb{B} \to `e \in \llbracket \ \mathsf{T} \ \rrbracket
    B⊆ base eb = eb
    \mathbb{B} \subseteq \llbracket \mathsf{S} \Rightarrow \mathsf{T} \rrbracket = \mathbb{T} \longrightarrow \mathbb{B} \subseteq \llbracket \mathsf{S} \Rightarrow \mathsf{T} \rrbracket \circ \mathbb{B} \subseteq \mathbb{T} \longrightarrow \mathbb{B}
     - Semantic typing implies a normal form can be read
     \llbracket \_ \rrbracket \subseteq \mathbb{T} : \forall \ \mathsf{T} \to \mathsf{d} \in \llbracket \ \mathsf{T} \ \rrbracket \to \mathsf{d} \in \mathbb{T}
     \llbracket \text{ base } \rrbracket \subseteq \mathbb{T} = \mathbb{B} \subseteq \mathbb{T}
     \llbracket \ S \Rightarrow T \ \rrbracket \subseteq \mathbb{T} = \mathbb{B} {\longrightarrow} \mathbb{T} \subseteq \mathbb{T} \circ \llbracket \ S \Rightarrow T \ \rrbracket \subseteq \mathbb{B} {\longrightarrow} \mathbb{T}
```

Figure 12. Semantic types inhabit the candidate space

```
\begin{split} |\_| : \mathsf{Ctx} \to \mathbb{N} \\ | \varnothing | &= \mathsf{zero} \\ | \varGamma : \_| &= \mathsf{suc} | \varGamma | \\ | &= \mathsf{env}|_{-} : \forall \varGamma \to \varGamma \models \mathsf{env} | \varGamma | \\ | &= \mathsf{env}|_{-} : \forall \varGamma \to \varGamma \models \mathsf{env} | \varGamma | \\ | &= \mathsf{env}|_{-} | \varGamma \{x\} \{T\}_{-} = \\ | &= \mathbb{B} \subseteq \llbracket \mathsf{T} \rrbracket (|\mathsf{v}| \in \mathbb{B} \ (\mathsf{idx} \to |\mathsf{v}| \ \mathsf{x} \mid \varGamma \mid))) \\ | &= \mathsf{normalization} : \varGamma \vdash \mathsf{t} :: \mathsf{T} \to \exists \llbracket \mathsf{v} \rrbracket \mathsf{t} \ \mathsf{has-normal-form} < | \varGamma \mid > \mathsf{v} \\ | &= \mathsf{normalization} \{\varGamma \} \{\mathsf{T} = \mathsf{T}\} \vdash \mathsf{t} \\ | &= \mathsf{vith} \ \mathsf{fundamental-lemma} \vdash \mathsf{t} \vdash \mathsf{env} | \varGamma \mid \\ | &= \mathsf{u} \vdash \mathsf{v} \vdash \mathsf{v} \\ | &= \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \\ | &= \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \\ | &= \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \\ | &= \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \\ | &= \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \\ | &= \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \\ | &= \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \vdash \mathsf{v} \\ | &= \mathsf{v} \vdash \mathsf{v} \\ | &= \mathsf{v} \vdash \mathsf{v} \vdash
```

Figure 13. Normalization of STLC

typed lambda calculus is total. However, in the case of full reduction, we use the evaluation relation more as a tool to obtain the normal form than a semantic definition. In this sense, our proof is more algorithmic, and indeed there are parts of the algorithm that may be incorrect, such as the conversion between de Brujin indices and de Brujin levels. The proof does not verify that this conversion is correct, only that it terminates. Showing that normalization is correct requires a separate proof.

For this reason, in future work we would like to show how we can additionally prove that the technique of normalization by evaluation is both complete and sound with respect to an equivalence model for the simply typed lambda calculus. The completeness property states that if two terms are equivalent, they have the same normal form. Soundness is that a term should be equivalent to its normal form. Proving these two properties shows us the correctness of of normalization by evaluation.

Abel proves that the technique is both complete and sound with respect to $\beta\eta$ -equivalence. Doing so requires extending normalization by evaluation to include explicit substitutions and η -expansions (for unique η -long forms). After adding these extensions, the proof employs a Kripke logical relation, used to reasons about future extended contexts.

In the case of our version, we would want to show that using evaluation to prove normalization with full reduction is complete and sound with respect to β -equivalence (as we are not computing η -long forms). Doing so, however, would naturally involve reasoning about substitutions, which we have explicitly avoided so far.

Instead, we would consider a different model of equivalence: denotational equivalence. Figure 14 shows a denotational semantics of STLC terms that interprets them as Agda values. Using this model, we wish to show the soundness of normalization by proving that a term t is denotationally equivalent to its normal form $v: t \in \mathbb{Z}$ v.

We have proven that normalization using weak head reduction is sound. This proof appears in appendix A. In this proof, we must reason about substitution as we essentially implement the operation (which we name close) to perform the delayed substitution represented by a closure's saved environment.

However, we cannot prove completeness as weak head reduction does not reduce under abstractions. Consider the terms $\lambda x.x$ and $\lambda x.(\lambda y.y)x$, which are denotationally equivalent but do not have the same normal form; both terms are already in their normal form.

In future work, we would like to extend this proof to show the soundness of proving normalization under full reduction by evaluation. We conjecture that we may not need to reason about substitutions to prove this result, as in going from weak head reduction to full reduction we were able to avoid having to implement the operation. This would make

```
- Denotation of types
\mathcal{T}[\![ ]\!]: \mathsf{Type} \to \mathsf{Set}
\mathcal{T} bool = Bool - Agda's boolean type
\mathcal{T} \llbracket \ \mathsf{S} \Rightarrow \mathsf{T} \ \rrbracket = \mathcal{T} \llbracket \ \mathsf{S} \ \rrbracket \to \mathcal{T} \llbracket \ \mathsf{T} \ \rrbracket
- Denotation of contexts
C[\![]\!]: Ctx \to Set
C[\![\Gamma]\!] = \forall \{T\} \to \Gamma \ni T \to \mathcal{T}[\![T]\!]
- Extending context denotations
\_\&\_: C\llbracket \ \Gamma \ \rrbracket \to \mathcal{T}\llbracket \ \mathsf{T} \ \rrbracket \to C\llbracket \ \Gamma :: \mathsf{T} \ \rrbracket
(\rho \& z) zero = z
(\rho \& z) (suc x) = \rho x
- Denotation of terms
\mathcal{E}[\![\ ]\!]: \Gamma \vdash \mathsf{T} \to C[\![\ \Gamma\ ]\!] \to \mathcal{T}[\![\ \mathsf{T}\ ]\!]
\mathcal{E} true \rho = \text{true}
\mathcal{E} false \rho = \text{false}
\mathcal{E} \| \text{var } \mathbf{x} \| \rho = \rho \mathbf{x}
\mathcal{E}[\![ \lambda t ]\!] \rho = \lambda z \rightarrow \mathcal{E}[\![ t ]\!] (\rho \& z)
\mathcal{E}[\![\![ \, \mathbf{r} \cdot \mathbf{s} \, ]\!]\!] \rho = \mathcal{E}[\![\![ \, \mathbf{r} \, ]\!]\!] \rho (\mathcal{E}[\![\![ \, \mathbf{s} \, ]\!]\!] \rho)
\mathcal{E}[\![ if r then s else t \![ \![ \![ \![ \![
     with \mathcal{E}[\![ r ]\!] \rho
... | true = \mathcal{E}[\![ s ]\!] \rho
... | false = \mathcal{E} \parallel t \parallel \rho
mutual
     - Denotation of environments
     G[\![ ]\!] : \operatorname{Env} \Gamma \to C[\![ \Gamma ]\!]
    G[\![ \gamma ]\!] X = D[\![ \gamma X ]\!]
     - Denotation of domain elements
     \mathcal{D}[\![\ ]\!]: \mathsf{Domain}\ \mathsf{T} \to \mathcal{T}[\![\ \mathsf{T}\ ]\!]
     \mathfrak{D} true \mathbb{I} = \text{true}
     \mathcal{D} false = false
     \mathcal{D} \llbracket \langle \lambda t \rangle \delta \rrbracket = \lambda z \rightarrow \mathcal{E} \llbracket t \rrbracket (\mathcal{G} \llbracket \delta \rrbracket \& z)
- Denotational equivalence
\mathcal{E} \equiv : \forall (\mathsf{t} \mathsf{v} : \Gamma \vdash \mathsf{T}) \to \mathsf{Set}
\_\mathcal{E} \equiv \_\{\Gamma\} \ \mathsf{t} \ \mathsf{v} = \forall \ \{\rho : C \llbracket \ \Gamma \ \rrbracket\} \to \mathcal{E} \llbracket \ \mathsf{t} \ \rrbracket \ \rho \equiv \mathcal{E} \llbracket \ \mathsf{v} \ \rrbracket \ \rho
```

Figure 14. Denotational semantics

for another valuable proof by logical relations that did not require substitution lemmas.

5 Related Work

Teaching proofs by logical relations. There are many tutorials that introduce the proof technique of logical relations. Ahmed [3] provides a general overview, discussing their use in proving several properties of programs. Harper [14] shows how Tait's method can can be used to prove normalization with weak head reduction of the simply typed

lambda calculus and later extends the technique to prove normalization with full reductions [15]. Skorstengaard [26] gives an in-depth introduction, going on to show how to prove properties more complex than normalization. Additionally, Abel et al. [2] provide a general tutorial for proving strong normalization by logical relations in their proposal. These tutorials continue to be excellent sources to familiarize oneself with the technique of proofs by logical relations. Despite this, they are tricky to adapt to a mechanized setting, as the technique of logical relations becomes obscured by the many substitution lemmas required. We provide an introduction to logical relations that is appropriate even when presented in a formalization. Cave and Pientka [8] also provide a mechanization of a proof of normalization with weak head reduction that does not suffer from substitution lemmas, though this is is because they mechanize the proof using Beluga [20]. Beluga is a programming language with direct support for reasoning about the binding of variables using higher-order abstract syntax [19], and substitution lemmas are handled automatically.

Normalization by evaluation. Martin-Löf [18] first demonstrates normalization by a semantic argument, later coined normalization by evaluation. Coquand and Dybjer [11] expand on this idea, observing that an algorithm for normalization by evaluation can be used for a proof of normalization in a more traditional sense. They prove normalization of STLC and additionally prove soundness of the technique. Although they formalize their results in ALF, the proof is presented in plain text for readability. With the capabilities of Agda for literate programming, and the increased popularity of mechanized reasoning, we are able to present our proofs in a formalized setting. Wieczorek and Biernacki [30] formalize Abel's complete proof of normalization by evaluation. They extend STLC with explicit substitutions to prove the soundness and completeness with respect to $\beta\eta$ equivalence. We do not use explicit substitutions, enabling us to draw a more direct comparison with our initial proof that evaluation is total. Sestini [25] shows how the technique of proving normalization with full reductions by evaluation can be modified to prove normalization with weak λ reduction. Weak λ -reduction is an extension of weak head reduction that allows for limited reductions in the bodies of abstractions, making the system confluent. Formalized in Agda, their result is related to our proof of normalization with weak head reduction in appendix A. We focus on weak head reduction here as our goal is to show a full proof by logical relations that can be used to introduce the technique.

Formalizing substitution lemmas. There are many options for alleviating the burden that is posed by substitution lemmas in a formalization. These range from the automation of such proofs [6, 27, etc.] to having these properties hold automatically, as Cave and Pientka [9] show is possible with Beluga. Specifically for Agda, Allais et al. [4] introduce the generic-syntax library for representing syntax

that can prove generic lemmas such as the fundamental lemma of logical relations. Although there has been significant progress in aiding the mechanization of proofs requiring substitution lemmas, it continues to be a topic of interest, with proofs by logical relations recently being proposed as a benchmark for mechanizing metatheory [2].

6 Conclusion

We have demonstrated that proofs by logical relations do not always need substitution lemmas. Consequently, the technique can be taught in a mechanized setting without being obscured by the many lemmas typically required. We prove that evaluation is total for the simply typed lambda calculus. Formalized in Agda, the proof by logical relations is remarkably short and remains as readable as it might in an informal setting. The result is a proof that makes for an appropriate introduction to the technique of logical relations, even when teaching programming language foundations through the use of a proof assistant as has become common.

We extend our proof to prove the more common property of normalization of the simply typed lambda calculus. We prove normalization with full reductions by formalizing the work of Abel [1]. The proof shows how we can establish richer properties of programs by logical relations while still avoiding substitution lemmas. We do not formally show that our proof of normalization with full reductions is correct, though refer to Wieczorek and Biernacki [30], who formalize Abel's full proof in Coq.

We hope that in presenting both proofs we are making logical relations more relatable in mechanized settings, and that future work can expand on how they can be used to prove other properties of programs with logical relations without needing substitution lemmas.

Acknowledgments

This work has been partially supported by the National Science Foundation under grant NSF 2006535.

References

- Andreas Abel. 2013. Normalization by evaluation: Dependent types and impredicativity. Habilitation. Ludwig-Maximilians-Universität München (2013).
- [2] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2019. POPLMark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming* 29 (2019), e19.
- [3] Amal Ahmed. 2013. Logical relations. Oregon Programming Languages Summer School (OPLSS) (2013).
- [4] Guillaume Allais, Robert Atkey, James Chapman, Conor McBride, and James McKinna. 2018. A type and scope safe universe of syntaxes with binding: their semantics and proofs. Proceedings of the ACM on Programming Languages 2, ICFP (2018), 1–30.
- [5] Thorsten Altenkirch. 1993. A formalization of the strong normalization proof for System F in LEGO. In *International Conference on Typed Lambda Calculi and Applications*. Springer, 13–28.

- [6] Brian Aydemir and Stephanie Weirich. 2010. LNgen: Tool support for locally nameless representations. (2010).
- [7] Ulrich Berger and Helmut Schwichtenberg. 1991. An inverse of the evaluation functional for typed lambda-calculus. (1991).
- [8] Andrew Cave and Brigitte Pientka. 2013. First-class substitutions in contextual type theory. In Proceedings of the Eighth ACM SIGPLAN international workshop on Logical frameworks & meta-languages: theory & practice. 15–24.
- [9] Andrew Cave and Brigitte Pientka. 2018. Mechanizing proofs with logical relations-Kripke-style. Mathematical structures in computer science 28, 9 (2018), 1606–1638.
- [10] Arthur Charguéraud. 2013. Pretty-big-step semantics. In European Symposium on Programming. Springer, 41–60.
- [11] Thierry Coquand and Peter Dybjer. 1997. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science* 7, 1 (1997), 75–94.
- [12] Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, Vol. 75. Elsevier, 381–392.
- [13] Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. Proofs and types. Vol. 7. Cambridge university press Cambridge.
- [14] Robert Harper. 2022. How to (Re)Invent Tait's Method. https://www.cs.cmu.edu/~rwh/courses/chtt/pdfs/tait.pdf
- [15] Robert Harper. 2022. Kripke-Style Logical Relations for Normalization. https://www.cs.cmu.edu/~rwh/courses/chtt/pdfs/kripke.pdf
- [16] Gilles Kahn. 1987. Natural semantics. In Annual symposium on theoretical aspects of computer science. Springer, 22–39.
- [17] Peter J Landin. 1964. The mechanical evaluation of expressions. The computer journal 6, 4 (1964), 308–320.
- [18] Per Martin-Löf. 1975. An intuitionistic theory of types: Predicative part. In Studies in Logic and the Foundations of Mathematics. Vol. 80. Elsevier, 73–118.
- [19] Frank Pfenning and Conal Elliott. 1988. Higher-order abstract syntax. *ACM sigplan notices* 23, 7 (1988), 199–208.
- [20] Brigitte Pientka and Jana Dunfield. 2010. Beluga: A framework for programming and reasoning with deductive systems (system description). In Automated Reasoning: 5th International Joint Conference, IJ-CAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings 5. Springer,
- [21] Benjamin C Pierce. 2002. Types and programming languages. MIT press.
- [22] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. 2023. Software Foundations. https://softwarefoundations.cis.upenn.edu
- [23] Gordon Plotkin. 1973. Lambda-definability and logical relations. Edinburgh University.
- [24] John C Reynolds. 1983. Types, abstraction and parametric polymorphism. In Information Processing 83, Proceedings of the IFIP 9th World Computer Congres. 513–523.
- [25] Filippo Sestini. 2019. Normalization by Evaluation for Typed Weak lambda-Reduction. In 24th International Conference on Types for Proofs and Programs (TYPES 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- [26] Lau Skorstengaard. 2019. An introduction to logical relations. arXiv preprint arXiv:1907.11133 (2019).
- [27] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 2019. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs. 166–180.
- [28] William W Tait. 1967. Intensional interpretations of functionals of finite type I. *The journal of symbolic logic* 32, 2 (1967), 198–212.

- [29] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2022. Programming Language Foundations in Agda. https://plfa.inf.ed.ac.uk/22.08/
- [30] Paweł Wieczorek and Dariusz Biernacki. 2018. A Coq formalization of normalization by evaluation for Martin-Löf type theory. In Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. 266–279.

A Extending §2: Normalization With Weak Head Reduction

We show in this section how to extend our proof of the totality of evaluation for the simply typed lambda calculus in 2 to prove normalization with weak head reduction. Most of the content is identical to the figures presented in the paper, we only extend it to show that we can read back a normal form from a term that is well-typed.

We additionally show that the normal form that we read back is denotationally equivalent to the original term, thereby proving the soundness of the technique.

A.1 Embedding of STLC in Agda (Figure 1)

```
- Types
data Type: Set where
   bool: Type
   \implies_: Type \rightarrow Type \rightarrow Type
variable S T : Type
- Typing contexts
data Ctx: Set where
   \emptyset: Ctx
   \cdot: : Ctx \rightarrow Type \rightarrow Ctx
variable \Gamma \Delta : Ctx
- Intrinsically scoped de Brujin indices
data \ni : Ctx \rightarrow Type \rightarrow Set where
   zero: \Gamma : T \ni T
   suc : \Gamma \ni T \rightarrow \Gamma :: S \ni T
variable x : \Gamma \ni T
- Intrinsically typed terms
data \vdash : Ctx \rightarrow Type \rightarrow Set where
   true false : \Gamma \vdash bool
   if then else : \Gamma \vdash \mathsf{bool} \to \Gamma \vdash \mathsf{T} \to \Gamma \vdash \mathsf{T} \to \Gamma \vdash \mathsf{T}
   var : \Gamma \ni T \rightarrow \Gamma \vdash T

\lambda : \Gamma : S \vdash T \rightarrow \Gamma \vdash S \Rightarrow T

   \cdot : \Gamma \vdash S \Rightarrow T \rightarrow \Gamma \vdash S \rightarrow \Gamma \vdash T
variable r s t u v : \Gamma \vdash T
```

A.2 Natural semantics (Figure 2)

```
Env \Gamma = \forall \{T\} \rightarrow \Gamma \ni T \rightarrow Domain T
    data Domain: Type → Set where
       true false: Domain bool
       - Closures
       \langle \lambda_{-} \rangle_{-} : \Gamma : S \vdash T \rightarrow \text{Env } \Gamma \rightarrow \text{Domain } (S \Rightarrow T)
variable \gamma \delta : \text{Env } \Gamma
variable a b d f : Domain T
\_++\_ : Env \Gamma \rightarrow Domain T \rightarrow Env (\Gamma :: T)
(++a) zero = a
(\gamma ++ \_) (suc x) = \gamma x
\mathsf{data} \ \_|\_ \Downarrow \_ : \mathsf{Env} \ \varGamma \to \varGamma \vdash \mathsf{T} \to \mathsf{Domain} \ \mathsf{T} \to \mathsf{Set} \ \mathsf{where}
   evalTrue : γ | true ↓ true
   evalFalse : y | false ↓ false
   evalVar : \gamma \mid \text{var } \mathbf{x} \Downarrow \gamma \mathbf{x}
   evalAbs : \gamma \mid \lambda t \downarrow \langle \lambda t \rangle \gamma
   evalApp:
       \gamma \mid r \downarrow \langle \lambda t \rangle \delta
        \rightarrow \gamma \mid s \parallel a
       \rightarrow \delta ++ a | t \downarrow b
       \rightarrow \gamma \mid r \cdot s \parallel b
   evallfTrue:
       \gamma \mid r \downarrow \text{true}
       \rightarrow \gamma \mid s \downarrow a
       \rightarrow \gamma | if r then s else t \downarrow a
   evallfFalse:
       y | r ↓ false
        \rightarrow \gamma \mid t \parallel b
       \rightarrow \gamma | if r then s else t \downarrow b
```

A.3 Proof by logical relations (Figure 3)

empty: Env Ø

This is the property we wish to prove, that evaluation is well-defined:

```
^{\land}: \Gamma \models \gamma \rightarrow a \in \llbracket T \rrbracket \rightarrow \Gamma : T \models \gamma ++ a
(⊧y ^ sa) zero = sa
(\models \gamma \land sa) (suc x) = \models \gamma x
semantic-type : \Gamma \vdash T \rightarrow Set
semantic-type \{\Gamma\} \{T\} t =
      \forall \{ \gamma : \mathsf{Env} \ \Gamma \} \to \Gamma \models \gamma
      \rightarrow \exists [a] y \mid t \downarrow a \times a \in [T]
syntax semantic-type \{\Gamma\} \{T\} t = \Gamma \models t :: T
fundamental-lemma : \forall (t : \Gamma \vdash T) \rightarrow \Gamma \models t :: T
fundamental-lemma true \( \psi \) = true, evalTrue, tt
fundamental-lemma false \neq y = \text{false}, evalFalse, tt
fundamental-lemma (var x) \{y\} \models y =
   \gamma x, evalVar, \models \gamma x
fundamental-lemma (\lambda t) \{ \gamma \} \neq \gamma =
      \langle \lambda t \rangle \gamma,
      evalAbs,
      \lambda sa \rightarrow fundamental-lemma t (\models \gamma ^ sa)
fundamental-lemma (r \cdot s) \models \gamma
   with fundamental-lemma r \models y
... |\langle \lambda t \rangle \delta, r\parallel, sf =
   let (a, s), (a, s) = fundamental-lemma (a, s) in
   let (b, t \downarrow l, sb) = sf sa in
   b, evalApp r \parallel s \parallel t \parallel, sb
fundamental-lemma (if r then s else t) \neq y
   with fundamental-lemma r ⊧y
... | true , r↓, _ =
   let (a, s \downarrow , sa) = fundamental-lemma s \models y in
   a, evallfTrue r↓s↓, sa
... | false , r↓, _ =
   let (b, t \downarrow \downarrow, sb) = fundamental-lemma t \models y in
   b, evallfFalse r \precept t \precept, sb
- Evaluation is total
\parallel-total : \forall (t : \varnothing \vdash T) \rightarrow \parallel-well-defined t
J-total t =
   let (a, t||a, ) = fundamental-lemma t (\lambda ()) in
   a,t∥a
A.4 Denotational semantics (Figure 14)
- Denotation of types
\mathcal{T}[\![ ]\!]: \mathsf{Type} \to \mathsf{Set}
\mathcal{T} bool = Bool
\mathcal{T} \llbracket \mathsf{S} \Rightarrow \mathsf{T} \rrbracket = \mathcal{T} \llbracket \mathsf{S} \rrbracket \to \mathcal{T} \llbracket \mathsf{T} \rrbracket
- Denotation of contexts
C \mathbb{I} : Ctx \to Set
C[\![\Gamma]\!] = \forall \{T\} \to \Gamma \ni T \to \mathcal{T}[\![T]\!]
```

– Extending denoted contexts $_\&_: C \llbracket \ \Gamma \ \rrbracket \to \mathcal{T} \llbracket \ \mathsf{T} \ \rrbracket \to C \llbracket \ \Gamma :: \mathsf{T} \ \rrbracket$

```
(& a) zero = a
(\rho \& \_) (suc x) = \rho x
- Denotation of terms
\mathcal{E}[\![\ ]\!]: \Gamma \vdash \mathsf{T} \to \mathcal{C}[\![\ \Gamma\ ]\!] \to \mathcal{T}[\![\ \mathsf{T}\ ]\!]
\mathcal{E} true \rho = \text{true}
\mathcal{E} false \rho = \text{false}
\mathcal{E}[\![\![ var x ]\!]\!] \rho = \rho x
\mathcal{E}[\![ \lambda t ]\!] \rho = \lambda a \rightarrow \mathcal{E}[\![ t ]\!] (\rho \& a)
\mathcal{E}[\![\![ \, \mathbf{r} \cdot \mathbf{s} \, ]\!]\!] \rho = \mathcal{E}[\![\![ \, \mathbf{r} \, ]\!]\!] \rho \, (\mathcal{E}[\![\![ \, \mathbf{s} \, ]\!]\!] \rho)
\mathcal{E} if r then s else t \rho
     with \mathcal{E}[\![ r ]\!] \rho
... | true = \mathcal{E}[\![ s ]\!] \rho
... | false = \mathcal{E} \llbracket t \rrbracket \rho
mutual
     - Denotation of environments
     G[\![ ]\!] : \operatorname{Env} \Gamma \to C[\![ \Gamma ]\!]
     G[\![ \gamma ]\!] x = D[\![ \gamma x ]\!]
     - Denotation of domain elements
     \mathcal{D}[\![ ]\!]: \mathsf{Domain} \ \mathsf{T} \to \mathcal{T}[\![ \ \mathsf{T} \ ]\!]
     \mathfrak{D} true = true
     \mathcal{D} false = false
     \mathcal{D}[\![\!] \langle \lambda \mathsf{t} \rangle \gamma ]\!] = \lambda \mathsf{a} \to \mathcal{E}[\![\!] \mathsf{t} ]\!] (\mathcal{G}[\![\!] \gamma ]\!] \& \mathsf{a})
- Denotational equivalence
\underline{\mathcal{E}} \equiv \underline{\phantom{}} : \forall \ (\mathsf{t} \ \mathsf{v} : \Gamma \vdash \mathsf{T}) \to \mathbf{Set}
\mathcal{E} = \{ \Gamma \} \mathsf{t} \mathsf{v} = \forall \{ \rho : C \llbracket \Gamma \rrbracket \} \to \mathcal{E} \llbracket \mathsf{t} \rrbracket \rho \equiv \mathcal{E} \llbracket \mathsf{v} \rrbracket \rho
```

A.5 Reading back a value from the evaluation of a term

So far, everything is the same as we have presented in the paper. Now, on top of proving that evaluation is total, we prove normalization with weak head reduction. To do so, we read back a value from the evaluation of a term. For this, we have to truly "close" a closure to produce a closed term that is a value.

```
inject true = true
inject false = false
inject (if t then u else v) =
   if inject t then inject u else inject v
- If we have a variable in a injected context
- we can determine where it came from
\mathsf{split}: \Gamma \mathrel{<\!\!\!>} \Delta \ni \mathsf{T} \to (\Gamma \ni \mathsf{T}) \uplus (\Delta \ni \mathsf{T})
split \{\Delta = \emptyset\} x = inj_1 x
split \{\Delta = \cdot : \} zero = inj<sub>2</sub> zero
split \{\Delta = \Delta : \_\} (suc x)
  with split \{\Delta = \Delta\} x
... | inj_1 x = inj_1 x
... | inj_2 y = inj_2 (suc y)
- Reading back a normal form from the evaluation of
- a term. We truly "close" a closure in reading it
- back to a normal form by replacing every variable
- in its context using its environment
mutual
   \uparrow: Domain T \rightarrow \emptyset \vdash T
  (\langle \lambda u \rangle \gamma) \uparrow = (\lambda (close \gamma u))
  true ↑ = true
   false ↑ = false
   - Note that this operation is essentially a
   substitution
   \mathsf{close} : (\mathsf{Env}\ \varGamma) \to \varGamma \mathrel{<>} \varDelta \vdash \mathsf{T} \to \varDelta \vdash \mathsf{T}
   close y true = true
   close y false = false
   close \{\Gamma = \Gamma\} \gamma (var x)
     with split \{\Gamma\} x
   ... | inj_2 y = var y
   ... | inj_1 \times with (\gamma \times) \uparrow
   ... | p = inject p
   close \gamma(\lambda t) = \lambda (close \gamma t)
   close \gamma (r · s) = close \gamma r · close \gamma s
  close \gamma (if t then u else v) =
     if close \gamma t then close \gamma u else close \gamma v
```

A.6 Normalization with weak head reduction

```
_normalizes-to_ : \varnothing \vdash T \to \varnothing \vdash T \to Set

t normalizes-to v = \exists [\ a\ ] \ empty \mid t \Downarrow a \times (a\ \Uparrow) \equiv v

normalization : (t : \varnothing \vdash T) \to \exists [\ v\ ] \ t normalizes-to v

normalization t

with fundamental-lemma t (\lambda\ ())

... |\ a\ , t \Downarrow\ , \_ = (a\ \Uparrow)\ , a\ , t \Downarrow\ , refl

- Normal form of a term is indeed a normal term

data Value : \varnothing \vdash T \to Set where
```

```
valueTrue : Value true valueFalse : Value false valueAbs : (t:\varnothing:S\vdash T)\to Value\ (\hbar\ t) \uparrow-Value : (a:Domain\ T)\to Value\ (a\ \uparrow) \uparrow-Value (\langle \hbar\ t\ \rangle\ \gamma)=valueAbs\ (close\ \gamma\ t) \uparrow-Value true = valueTrue \uparrow-Value false = valueFalse
```

A.7 Soundness

postulate

Now, we show that a term is in fact denotationally equivalent to its normal form as determined by our proof. To do so, we need to show that the normal form we read back is equivalent to the evaluation of the term (which we have already proven is equivalent to the term itself).

```
fext : Extensionality 0ℓ 0ℓ
fext-implicit = implicit-extensionality fext
denote-&-++: G \llbracket \delta \rrbracket \& \mathcal{D} \llbracket a \rrbracket \equiv G \llbracket \delta ++ a \rrbracket \{T\}
denote-&-++ =
   fext \lambda where zero \rightarrow refl; (suc ) \rightarrow refl
\Downarrow-sound : \gamma \mid t \Downarrow a \rightarrow \mathcal{E} \llbracket t \rrbracket \mathcal{G} \llbracket \gamma \rrbracket \equiv \mathcal{D} \llbracket a \rrbracket
 ∥-sound evalTrue = refl
 J-sound evalVar = refl
∥-sound evalAbs = refl
\downarrow -sound (evalApp {t = t} {\delta} {a = a} {b} r \downarrow  s \downarrow  t \downarrow )
   rewrite \Downarrow-sound r \Downarrow | \Downarrow-sound s \Downarrow =
   begin
      \mathcal{E}[\![ t ]\!] (\mathcal{G}[\![ \delta ]\!] \& \mathcal{D}[\![ a ]\!])
   \equiv \langle \operatorname{cong} \mathcal{E} | \mathsf{t} | | (\text{fext-implicit denote-} \& -++) \rangle
      \mathcal{E}[\![t]\!]\mathcal{G}[\![\delta++a]\!]
   \equiv \langle \downarrow \text{-sound } t \downarrow \rangle
       D b
\parallel-sound (evalIfTrue r \parallel s \parallel)
   rewrite \parallel-sound r \parallel \parallel \parallel-sound s \parallel = refl
↓-sound (evalIfFalse r↓ t↓)
   rewrite \parallel-sound r \parallel \mid \parallel-sound t \parallel = refl
- Natural semantics is deterministic
\Downarrow-uniq : \gamma \mid t \Downarrow a \rightarrow \gamma \mid t \Downarrow b \rightarrow a \equiv b
↓-uniq evalTrue evalTrue = refl
↓-uniq evalFalse evalFalse = refl
↓-uniq evalVar evalVar = refl
↓-uniq evalAbs evalAbs = refl
```

 \Downarrow -uniq (evalApp $r \Downarrow_1 s \Downarrow_1 t \Downarrow_1$) (evalApp $r \Downarrow_2 s \Downarrow_2 t \Downarrow_2$)

rewrite \parallel -unig $s \parallel_1 s \parallel_2 \parallel$ -unig $t \parallel_1 t \parallel_2 = refl$

with \parallel -uniq $r \parallel_1 r \parallel_2$

... | refl

```
\Downarrow-uniq (evallfTrue r \Downarrow_1 s \Downarrow_1) evalIf
   with evalIf
... | evallfTrue r \parallel_2 s \parallel_2
   rewrite \parallel-uniq r \parallel_1 r \parallel_2 \mid \parallel-uniq s \parallel_1 s \parallel_2 = refl
... | evallfFalse r \downarrow _2 =
   contradiction (\Downarrow-uniq r \Downarrow_1 r \Downarrow_2) (\lambda ())
\Downarrow-uniq (evalIfFalse r \Downarrow_1 t \Downarrow_1) evalIf
   with evalIf
... | evallfFalse r \parallel_2 t \parallel_2
   rewrite \Downarrow-uniq r \Downarrow_1 r \Downarrow_2 | \Downarrow-uniq t \Downarrow_1 t \Downarrow_2 = refl
... | evallfTrue r \downarrow_2 =
   contradiction (\parallel-uniq r \parallel_1 r \parallel_2) (\lambda ())
- Appending denoted contexts
\_\&\&\_: C\llbracket \Gamma \rrbracket \to C\llbracket \Delta \rrbracket \to C\llbracket \Gamma \Leftrightarrow \Delta \rrbracket
\&\& \{\Delta = \emptyset\} p q = p
\&\&_{\{\Delta = \Delta : T\}} p q = (p \&\& q \circ suc) \& q zero
- Evaluation of a term injected to an extended context
- is equivalent
&-inject:
   \forall (\mathsf{t} : \Gamma \vdash \mathsf{T}) (\mathsf{p} : C \llbracket \Gamma \rrbracket) (\mathsf{q} : C \llbracket \Delta \rrbracket)
   \rightarrow \mathcal{E}[\![t]\!] p \equiv \mathcal{E}[\![t]\!] inject t [\![t]\!] (q \&\& p)
\mathcal{E}-inject true p q = refl
\mathcal{E}-inject false p q = refl
&-inject (var zero) p q = refl
\mathcal{E}-inject (var (suc x)) p q
   rewrite \mathcal{E}-inject (var x) (p \circ suc) q = refl
\mathcal{E}-inject (\lambda t) p q =
   fext (\lambda x \rightarrow \mathcal{E}-inject t (p & x) q)
\mathcal{E}-inject (r \cdot s) p q
   rewrite \mathcal{E}-inject r p q | \mathcal{E}-inject s p q = refl
&-inject (if r then s else t) p q
   with \mathcal{E}[r] p | \mathcal{E}[inject r] (q && p) | \mathcal{E}-inject r p q
... | true | true | refl = \mathcal{E}-inject s p q
... | false | false | refl = &-inject t p q
- Splitting variable into left context
split-left:
   \forall \{x : \Gamma \Longleftrightarrow \Delta \ni T\} \{y : \Gamma \ni T\} \{p : C \llbracket \Gamma \rrbracket \} \{q : C \llbracket \Delta \rrbracket \}
   \rightarrow split x \equiv inj_1 y
   \rightarrow (p && q) x \equiv p y
split-left \{\Delta = \emptyset\} \{x = zero\} refl = refl
split-left \{\Delta = \emptyset\} \{x = suc x\} refl = refl
split-left \{\Delta = \Delta : S\} \{x = suc x\} eq
   with split \{\Delta = \Delta\} x in eq' | eq
\dots | inj_1 | refl =
   split-left \{\Delta = \Delta\} eq'
- Splitting variable into right context
split-right:
   \forall \{\mathsf{x}: \Gamma \Longleftrightarrow \Delta \ni \mathsf{T}\} \{\mathsf{y}: \Delta \ni \mathsf{T}\} \{\mathsf{p}: C \llbracket \Gamma \rrbracket\} \{\mathsf{q}: C \llbracket \Delta \rrbracket\}
```

```
\rightarrow split x \equiv inj_2 y \rightarrow (p \&\& q) x \equiv q y
split-right \{\Delta = \Delta : \bot\} \{x = zero\} refl = refl
split-right \{\Delta = \Delta : \_\} \{x = suc x\} eq
   with split \{\Delta = \Delta\} x in eq' | eq
... | inj<sub>2</sub> _ | refl =
   split-right \{\Delta = \Delta\} eq'
mutual
   - Evaluation of a term is equivalent to evaluation
   - of its closed self
   &-close:
      \forall (\mathsf{t} : \Gamma \iff \Delta \vdash \mathsf{T}) (\gamma : \mathsf{Env} \ \Gamma) (\mathsf{q} : C \llbracket \Delta \rrbracket)
      \rightarrow \mathcal{E}[\![t]\!] (\mathcal{G}[\![\gamma]\!] \&\& q) \equiv \mathcal{E}[\![close\ \gamma\ t]\!] q
   &-close true _ = refl
   \mathcal{E}-close false = refl
   \mathcal{E}-close \{\Delta = \Delta\} (var x) \gamma q
      with split \{\Delta = \Delta\} x in eq
   ... | inj_2 y = split-right \{p = G [ \gamma ] \} eq
   ... | inj<sub>1</sub> x'
      rewrite split-left \{p = G \| y \|\} \{q = q\} eq
      rewrite \uparrow-sound \{\rho = \lambda ()\} (\gamma x') =
      \mathcal{E}-inject ((\gamma x') ↑) (\lambda ()) q
   \mathcal{E}-close (\mathcal{T} t) \gamma q =
      fext (\lambda x \rightarrow \mathcal{E}-close t \gamma (q & x))
   \mathcal{E}-close (\mathbf{r} \cdot \mathbf{s}) \gamma \mathbf{q}
      rewrite ε-close r γ q
          |\mathcal{E}-close s \gamma q = refl
   \mathcal{E}-close (if r then s else t) \gamma q
      with \mathcal{E}[\![r]\!] (\mathcal{G}[\![\gamma]\!] && q) | \mathcal{E}[\![close\ \gamma\ r]\!] q | \mathcal{E}-close r \gamma q
   ... | true | true | refl = \mathcal{E}-close s \gamma q
   ... | false | false | refl = \mathcal{E}-close t \gamma q
   - Reading back a normal form from an evaluated term
   - preserves meaning
   ↑-sound:
      \forall \{ \rho : C \llbracket \varnothing \rrbracket \} (a : Domain T)
      \rightarrow \mathcal{D} \llbracket a \rrbracket \equiv \mathcal{E} \llbracket a \uparrow \rrbracket \rho
   \uparrow-sound \{\rho = \rho\} (\langle \hat{\chi}_{-} \rangle_{-} \{\Gamma\} \mathsf{t} \gamma)
      = fext (\lambda a \rightarrow \mathcal{E}-close t \gamma (\rho & a))
   ↑-sound true = refl
   ↑-sound false = refl
- Use the fact that reading back a normal form is sound
- w.r.t. denotational semantics to prove normalization
- is sound
normalization-sound:
   \forall (t : \emptyset \vdash T) (v : \emptyset \vdash T)
   → t normalizes-to v
   \rightarrow t \mathcal{E} \equiv v
normalization-sound t (,t), refl\{\rho\}
   with normalization t
```

```
... | v, a, t | ', refl

with | -uniq t | t | t | '

... | refl =

begin

\mathcal{E} [ t ] \rho

\equiv \langle cong \mathcal{E} [ t ] (fext-implicit (fext <math>\lambda ())) \rangle

\mathcal{E} [ t ] \mathcal{G} [ empty ] ]

\equiv \langle | -sound t | \rangle

\mathcal{D} [ a ] ]

\equiv \langle | -sound a \rangle

\mathcal{E} [ v ] \rho
```

B Normalization with full reductions (full proof of §3)

Here we provide the full proof presented in §3, including any portions that were omitted in the main body of the paper.

B.1 STLC (Figure 4)

```
data Type: Set where
   base: Type
   \Rightarrow_: Type \rightarrow Type \rightarrow Type
variable S T : Type
data Ctx: Set where
   \emptyset: Ctx
   : Ctx \rightarrow Type \rightarrow Ctx
variable \Gamma: Ctx
- Raw terms
data Term: Set where
   var: (x: \mathbb{N}) \to Term

\lambda_{-}: \mathsf{Term} \to \mathsf{Term}

   \_\cdot\_: \mathsf{Term} \to \mathsf{Term} \to \mathsf{Term}
variable x : N
variable r s t u v : Term

    Variable lookup

data \_:: _{\in} : \mathbb{N} \to \mathsf{Type} \to \mathsf{Ctx} \to \mathsf{Set} where
   here : zero :: T \in \Gamma ·: T
   there : x :: T \in \Gamma \rightarrow suc \ x :: T \in \Gamma :: S
- Typing judgement
data \_\vdash_::_ : Ctx \rightarrow Term \rightarrow Type \rightarrow Set where
   \vdashvar : x :: T \in \Gamma \rightarrow \Gamma \vdash var x :: T
   \vdashabs : \Gamma :: S \vdash t :: T \rightarrow \Gamma \vdash \lambda t :: S \Rightarrow T
   \vdashapp: \Gamma \vdash r :: S \Rightarrow T \rightarrow \Gamma \vdash s :: S \rightarrow \Gamma \vdash r \cdot s :: T
```

B.2 Natural semantics (Figure 6)

```
mutual
   - Environments
   Env = \mathbb{N} \rightarrow Domain
   - Domain
   data Domain: Set where
      \langle \lambda \rangle: Term \rightarrow Env \rightarrow Domain
       ": Domain ^{ne} \rightarrow Domain
   - Neutral domain
   data Domain<sup>ne</sup>: Set where
      |v|:(k:\mathbb{N})\to Domain^{ne}
      \_\cdot\_: \mathsf{Domain}^{ne} \to \mathsf{Domain} \to \mathsf{Domain}^{ne}
variable \gamma: Env
variable a b d f : Domain
variable e : Domain<sup>ne</sup>
variable k : \mathbb{N} - de Brujin level
\_++\_ : Env \rightarrow Domain \rightarrow Env
(\underline{\phantom{a}} ++ a) zero = a
(\gamma ++ \_) (suc m) = \gamma m
- Natural semantics
mutual
   data \parallel \downarrow \parallel : Env \rightarrow Term \rightarrow Domain \rightarrow Set where
      evalVar : \gamma \mid \text{var } \mathbf{x} \downarrow \gamma \mathbf{x}
      evalAbs : \gamma \mid \lambda t \downarrow \langle \lambda t \rangle \gamma
      evalApp: \gamma \mid r \downarrow f \rightarrow \gamma \mid s \downarrow a \rightarrow f \cdot a \downarrow b \rightarrow \gamma \mid r \cdot s \downarrow b
   data \cdot \downarrow : Domain \rightarrow Domain \rightarrow Set where
      appClosure : \gamma ++ a \mid t \parallel b \rightarrow \langle \lambda t \rangle \gamma \cdot a \parallel b
      appNeutral: e \cdot d \parallel (e \cdot d)
```

B.3 Reading a normal form back from an evaluated term (Figure 7)

```
variable n: \mathbb{N} - Scope

- Converting a de Brujin level to a de Brujin index |v| \rightarrow idx : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
|v| \rightarrow idx : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
|v| \rightarrow idx : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
|v| \rightarrow idx : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
|v| \rightarrow idx : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
|v| \rightarrow idx : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
|v| \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
|v| \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
|v| \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
|v| \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
```

B.4 Asking more of semantic types with candidate spaces (Figure 8)

```
LogPred = Domain \rightarrow Set
- Bottom of candidate space
- (neutral term can be read back)
\mathbb{B}: \mathsf{LogPred}
\mathbb{B} ('e) = \forall n \rightarrow \exists [u] n | e \uparrow^{ne} u
\mathbb{B}_{-} = \bot
- Top of candidate space
- (normal term can be read back)
\mathbb{T}: LogPred
\mathbb{T} d = \forall n \rightarrow \exists [v] n \mid d \uparrow v
- Bottom of space is a subset of top
\mathbb{B} {\subseteq} \mathbb{T} : d \in \mathbb{B} \to d \in \mathbb{T}
\mathbb{B} \subseteq \mathbb{T} \{ (e) \}  eb n
  with eb n
... | u , eîu =
   u, neutral enu
```

B.5 Proof by logical relations

```
\longrightarrow : LogPred \rightarrow LogPred \rightarrow LogPred
(A \longrightarrow B) f = \forall \{a\} \longrightarrow a \in A \longrightarrow \exists [b] f \cdot a \parallel b \times b \in B
[\![ ]\!]: \mathsf{Type} \to \mathsf{LogPred}
a base a = B
\llbracket S \Rightarrow T \rrbracket = \llbracket S \rrbracket \longrightarrow \llbracket T \rrbracket
_{\models}: Ctx \rightarrow Env \rightarrow Set
\Gamma \models \gamma = \forall \{x\} \{T\} \rightarrow x :: T \in \Gamma \rightarrow \gamma x \in [T]
\models :: : Ctx \rightarrow Term \rightarrow Type \rightarrow Set
\Gamma \models \mathsf{t} :: \mathsf{T} = \forall \{\gamma\} \to \Gamma \models \gamma \to \exists [\mathsf{a}] \gamma \mid \mathsf{t} \Downarrow \mathsf{a} \times \mathsf{a} \in [\![\mathsf{T}]\!]
^{\wedge}: \Gamma \models \gamma \rightarrow a \in [S] \rightarrow \Gamma : S \models \gamma ++ a
    ^_ _ sa here = sa
_{^{\prime}} \models \gamma  (there pf) = \models \gamma pf
fundamental-lemma : \Gamma \vdash t :: T \rightarrow \Gamma \models t :: T
fundamental-lemma (\vdashvar {x} pf) {\gamma} \models \gamma =
    y x, evalVar, ⊧y pf
fundamental-lemma \{t = \lambda t\} (+abs +t) \{y\} \neq y = \lambda t
    \langle \lambda t \rangle \gamma,
    evalAbs,
   \lambda sa \rightarrow
```

```
let \modelst = fundamental-lemma \modelst in let (b, eval-closure, sb) = \modelst (\modelsy ^ sa) in b, appClosure eval-closure, sb fundamental-lemma (\modelsapp \modelsr \modelss) \modelsy = let (f, r\Downarrow, sf) = fundamental-lemma \modelsr \modelsy in let (a, s\Downarrow, sa) = fundamental-lemma \modelss \modelsy in let (b, app-eval, sb) = sf sa in b, evalApp r\Downarrow s\Downarrow app-eval, sb
```

B.6 Semantically typed domain elements inhabit the candidate space

```
\mathbb{B} \subseteq \mathbb{T} \longrightarrow \mathbb{B} : `e \in \mathbb{B} \rightarrow `e \in \mathbb{T} \longrightarrow \mathbb{B}
\mathbb{B} \subseteq \mathbb{T} \longrightarrow \mathbb{B} \{e\} \ eb \{d\} \ dt =
    (e \cdot d),
    appNeutral,
    \lambda n \rightarrow
         let (u, e↑u) = eb n in
         let (v, d \uparrow v) = dt n in
         u · v , ↑app e↑u d↑v
|\mathbf{v}| \in \mathbb{B} : \forall \ k \rightarrow \text{`lvl} \ k \in \mathbb{B}
|v| \in \mathbb{B} k n = var(|v| \rightarrow idx k n), \uparrow |v|
\mathbb{B} {\longrightarrow} \mathbb{T} {\subseteq} \mathbb{T} : d \in \mathbb{B} {\longrightarrow} \mathbb{T} {\longrightarrow} d \in \mathbb{T}
\mathbb{B} \longrightarrow \mathbb{T} \subseteq \mathbb{T} \{ \langle \lambda t \rangle \gamma \} \text{ pf n}
    with pf (|v| \in \mathbb{B} n)
... | d, appClosure eval-closure, dt
    with dt n
... | v, d \uparrow v =
    \mathbb{B} \longrightarrow \mathbb{T} \subseteq \mathbb{T} \{ e \} pf n
    with pf (lvl∈B n)
... | _ , appNeutral , et
    with et n
... | \mathbf{u} \cdot \mathbf{v} | (\( \frac{1}{2} \text{app e} \)\( \frac{1}{2} \text{u} \) =
    u, ↑neutral e↑u
mutual
    \mathbb{T} {\longrightarrow} \mathbb{B} {\subseteq} \llbracket \_ {\Rightarrow} \_ \rrbracket : \forall \ S \ T \to f \in \mathbb{T} \longrightarrow \mathbb{B} \to f \in \llbracket \ S \Rightarrow T \ \rrbracket
    \mathbb{T} \longrightarrow \mathbb{B} \subseteq [\![ S \Rightarrow T ]\!] \text{ pf sa}
         with ¶S ∥⊆T sa
    ... | at
         with pf at
    ... | 'e, app-eval, eb
         with \mathbb{B}\subseteq \llbracket \mathsf{T} \rrbracket eb
    ... | se =
          'e,app-eval,se
    \llbracket \_ \Rightarrow \_ \rrbracket \subseteq \mathbb{B} \longrightarrow \mathbb{T} : \forall \ S \ T \rightarrow f \in \llbracket \ S \Rightarrow T \ \rrbracket \rightarrow f \in \mathbb{B} \longrightarrow \mathbb{T}
    \llbracket S \Rightarrow T \rrbracket \subseteq \mathbb{B} \longrightarrow \mathbb{T} \text{ sf } \{\text{`e}\} \text{ eb}
         with sf (\mathbb{B}\subseteq [S] eb)
```

```
\begin{split} ... & \mid d \text{ , app-eval , } sd = \\ & d \text{ , app-eval , } \llbracket T \rrbracket \subseteq \mathbb{T} \text{ sd} \\ - & \text{Any object that can have a neutral form read back } \\ - & \text{ is semantically typed} \\ & \mathbb{B} \subseteq \llbracket \rrbracket : \forall \ T \to `e \in \mathbb{B} \to `e \in \llbracket T \rrbracket \\ & \mathbb{B} \subseteq \llbracket \text{ base } \rrbracket \text{ eb} = \text{eb} \\ & \mathbb{B} \subseteq \llbracket S \Rightarrow T \rrbracket = \mathbb{T} \longrightarrow \mathbb{B} \subseteq \llbracket S \Rightarrow T \rrbracket \circ \mathbb{B} \subseteq \mathbb{T} \longrightarrow \mathbb{B} \\ - & \text{ Semantic typing implies a normal form can be read } \\ - & \text{ back } \\ & \mathbb{L} \rrbracket \subseteq \mathbb{T} : \forall \ T \to d \in \llbracket T \rrbracket \to d \in \mathbb{T} \\ & \mathbb{B} \text{ base } \mathbb{J} \subseteq \mathbb{T} = \mathbb{B} \longrightarrow \mathbb{T} \subseteq \mathbb{T} \circ \llbracket S \Rightarrow T \rrbracket \subseteq \mathbb{B} \longrightarrow \mathbb{T} \end{split}
```

B.7 Establishing normalization (Figure 13)

```
- Normalization of STLC
idx \rightarrow lvl : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
idx \rightarrow |v| i n = n - suc i
env : \mathbb{N} \to Env
env n i = 'lvl(idx \rightarrow lvl i n)
_{\text{has-normal-form}}: Term \rightarrow \mathbb{N} \rightarrow \text{Term} \rightarrow \text{Set}
t has-normal-form< n > v =
    \exists [a] env n | t \downarrow a \times n | a \uparrow v
|\_|: \mathsf{Ctx} \to \mathbb{N}
|\emptyset| = zero
|\Gamma : \_| = suc |\Gamma|
\models env | \_| : \forall \Gamma \rightarrow \Gamma \models env | \Gamma |
\models env \mid \Gamma \{x\} \{T\} =
   \mathbb{B}\subseteq \llbracket \mathsf{T} \rrbracket (\mathsf{Ivl}\in \mathbb{B} (\mathsf{idx} \rightarrow \mathsf{Ivl} \mathsf{x} \mid \Gamma \mid))
normalization : \Gamma \vdash t :: T \rightarrow \exists [v] t \text{ has-normal-form} < |\Gamma| > v
normalization \{\Gamma\}\{T = T\} + t
    with fundamental-lemma ⊢t ⊧env| Γ |
... | a, t | la, st
   with \llbracket T \rrbracket \subseteq \mathbb{T} st | \Gamma |
... | v , a∏v =
   v,a,t∥a,aîv
```

B.8 Read back produces a normal term

```
    Normal and neutral terms
    mutual
    data Normal : Term → Set where
    normalAbs : Normal t → Normal (ħt)
    neutral : Neutral t → Normal t
    data Neutral : Term → Set where
    neutralVar : Neutral (var x)
    neutralApp : Neutral r → Normal s → Neutral (r · s)
```

C Totality of evaluation: extrinsic representation using raw bindings

Here we provide a proof of that evaluation of the simply typed lambda calculus is total using an extrinsic representation of terms and raw variable bindings.

Using raw variable bindings in a mechanized proof by logical relations is generally not worth the readability it may add as proving substitution lemmas becomes increasingly cumbersome. As we do not need any such lemmas, our proof remains nearly as short as before.

In the main body of the paper, we this proof using an intrinsically typed representation as it simplifies a proof of soundness with respect to a denotational semantics when we extend the proof to show weak head normalization. As a result, it is easier to discuss this result in §4.

C.1 Embedding of STLC in Agda

```
variable x y : Id
Types
data Type : Set where
base : Type
⇒ : Type → Type → Type
variable S T : Type
Contexts
data Ctx : Set where
Ø : Ctx
, :: : Ctx → Id → Type → Ctx
variable Γ Δ : Ctx
Terms
data Term : Set where
var : Id → Term
λ → : Id → Term → Term
```

```
variable r s t : Term

- Variable lookup judgement into a context data _::_ ∈_ : Id → Type → Ctx → Set where here : x :: T ∈ \Gamma , x :: T there : x :: T ∈ \Gamma , \Gamma :: T there : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : x :: T ∈ \Gamma := Term → Type → Set where evar : T ∈ \Gamma := Term → Type → Set where evar : T ∈ \Gamma := Term → Type → Set where evar : T ∈ \Gamma := Term → Type → Set where evar : T ∈ \Gamma := Term → Type → Set where evar : T ∈ \Gamma := Term → Type → Set where evar : T ∈ \Gamma := Term → Type → Set where evar : T ∈ \Gamma := Term → Type → T
```

C.2 Natural semantics

- Environments + domains

```
mutual
   data Env: Set where
       Ø:Env
       \_,\_\mapsto\_: \mathsf{Env} \to \mathsf{Id} \to \mathsf{Domain} \to \mathsf{Env}
   data Domain: Set where
       \langle \lambda \longrightarrow \rangle : Id \rightarrow Term \rightarrow Env \rightarrow Domain
variable \gamma \delta: Env
variable a b f : Domain
- Variable lookup judgement into an environment
data \mapsto \in : Id \to Domain \to Env \to Set where
   here : x \mapsto a \in \gamma, x \mapsto a
   there : x \mapsto a \in \gamma \rightarrow \neg (x \equiv y) \rightarrow x \mapsto a \in \gamma, y \mapsto b
- Natural semantics
data \mid \downarrow \downarrow : Env \rightarrow Term \rightarrow Domain \rightarrow Set where
   evalVar : x \mapsto a \in \gamma \rightarrow \gamma \mid var x \downarrow a
   evalAbs: \gamma \mid \lambda x \longrightarrow t \downarrow \langle \lambda x \longrightarrow t \rangle \gamma
   evalApp: \gamma \mid r \Downarrow \langle \lambda x \longrightarrow t \rangle \delta
       \rightarrow \gamma \mid s \downarrow a
       \rightarrow \delta, x \mapsto a | t \downarrow b
       \rightarrow \gamma \mid \mathbf{r} \cdot \mathbf{s} \parallel \mathbf{b}
```

C.3 Proof by logical relations

```
- Semantic Types

[_]: Type → (Domain → Set)

[ base ] _ = ⊥

[ S ⇒ T ] (\langle \lambda x \longrightarrow t \rangle \delta) =

∀ {a: Domain}

→ a ∈ [ S ]

→ ∃[ b ] \delta , x \mapsto a | t | b × b ∈ [ T ]

- Semantic typing for environments
```

```
\models : Ctx \rightarrow Env \rightarrow Set
\Gamma \models \gamma = \forall \{x : Id\} \{T : Type\}
              \rightarrow x :: T \in \Gamma
             \rightarrow \exists [a] (x \mapsto a \in \gamma) \times a \in [T]
- Extending semantically typed environments
^{\land}: \Gamma \models \gamma \rightarrow a \in \llbracket T \rrbracket \rightarrow \Gamma, x :: T \models \gamma, x \mapsto a
_{a = a} \{x = x\}  sa here = a , here , sa
_{^{\prime}} \models \gamma  (there x \in \Gamma x \not\equiv y) =
   let (b, x \in \gamma, sb) = \models \gamma x \in \Gamma in
   b, there x \in \gamma x \not\equiv y, sb
- Semantic typing for terms
_{\models}::_ : Ctx \rightarrow Term \rightarrow Type \rightarrow Set
\Gamma \models \mathsf{t} :: \mathsf{T} = \forall \{ \gamma : \mathsf{Env} \}
                    \rightarrow \Gamma \models \gamma
                    \rightarrow \exists [a] \gamma \mid t \Downarrow a \times a \in [T]
- Well-typed terms are semantically typed
fundamental-lemma : \Gamma \vdash t :: T \rightarrow \Gamma \models t :: T
fundamental-lemma (\vdashvar x \in \Gamma) \models \gamma =
   let (a, x \in \gamma, sa) = \models \gamma x \in \Gamma in
```

```
a, evalVar x∈y, sa
fundamental-lemma \{t = \lambda x \longrightarrow t\} (\vdash abs \vdash t) \{\gamma\} \models \gamma = t\}
  \langle \lambda x \longrightarrow t \rangle \gamma,
  evalAbs,
  \lambda sa \rightarrow fundamental-lemma \vdasht (\models\gamma ^ sa)
fundamental-lemma (\vdashapp \vdashr \vdashs) \modelsy
  with fundamental-lemma ⊢r ⊨y
... |\langle \lambda x \longrightarrow t \rangle \delta, r \parallel, sf =
  let (a, s \downarrow , sa) = fundamental-lemma \vdash s \models y in
  let (b, f \downarrow , sb) = sf sa in
  b, evalApp r \parallel s \parallel f \parallel, sb
- Arbitrary environment as environments are no longer
- scoped by a context in extrinsic representation
variable empty : Env
- Totality of evaluation for well-typed terms in
- well-typed environments
\downarrow \text{-total} : \varnothing \vdash t :: T \rightarrow \exists [a] \text{ empty} \mid t \downarrow a
J-total ⊢t =
  let (a, t \parallel a,  ) = fundamental-lemma  + t (\lambda ()) in
  a,t∜a
```