JFP, 33 pages, 2025.
doi:10.1017/

THEORETICAL PEARL

Functional Pearl: Short and Mechanized Logical Relation for Dependent Type Theories

YIYUN LIU

University of Pennsylvania (e-mail: liuyiyun@seas.upenn.edu)

JONATHAN CHAN

University of Pennsylvania (e-mail: jcxz@seas.upenn.edu)

STEPHANIE WEIRICH

University of Pennsylvania (e-mail: sweirich@seas.upenn.edu)

Abstract

Proof by logical relations is a powerful technique that has been used to derive metatheoretic properties of type systems, such as consistency and parametricity. While there exists a plethora of introductory materials about logical relations in the context of simply typed or polymorphic lambda calculi, a streamlined presentation of proof by logical relation for a dependently typed language is lacking. In this paper, we present a short consistency proof for a dependently typed language that contains a rich set of features, including a countable universe hierarchy, booleans, and an intensional identity type. We show that the logical relation can be easily extended to prove the existence of $\beta\eta$-normal forms. We have fully mechanized the consistency proof using the Coq proof assistant in under 1000 lines of code, with 500 lines of additional code for the $\beta\eta$-normal form extension.

1 Introduction

This paper presents a *short* and *mechanized* proof of logical consistency for λ^{Π}, a dependent type theory with a full predicative universe hierarchy, large eliminations, an intensional identity type, a boolean base type, and dependent elimination forms.

Our goal with this work is to demonstrate the application of the proof technique of *syntactic logical relations* to dependent type theories. Logical relations are a powerful proof technique, and have been used to show diverse properties such as strong normalization (Girard et al., 1989; Geuvers, 1994), contextual equivalence (Constable et al., 1986), representation independence (Pitts, 1998), noninterference (Bowman and Ahmed, 2015), compiler correctness (Benton and Hur, 2009; Perconti and Ahmed, 2014), and the decidability of conversion algorithms (Harper and Pfenning, 2005; Abel and Scherer, 2012; Abel, 2013).

However, tutorial material on syntactic logical relations (Skorstengaard, 2019; Harper, 2022*a*,b; Pierce, 2002, 2004; Harper, 2016) is primarily focused on systems with simple or polymorphic types. In that context, syntactic logical relations can be defined as simple recursive functions over the structure of types, or (in the case of recursive types) defined over the evaluation steps of the computation. Yet, neither of these techniques can be used to define a logical relation in the context of a predicative dependent type theory, so a novice researcher might be excused for thinking that proofs that use logical relations are not applicable for such languages.

But this is not the case. Recent authors have developed tour-de-force mechanizations for the metatheory of modern proof assistants (Wieczorek and Biernacki, 2018; Abel et al., 2017; Adjedj et al., 2024; Anand and Rahli, 2014), and have relied on logical relations defined as part of their developments. However, because these proofs show diverse results about real systems and algorithms, these developments range in size from 10,000 to 300,000 lines of code. As a result, their uses of logical relations are difficult to isolate from the surrounding contexts and inaccessible to casual readers.

Thus, our paper provides a gentle and accessible introduction to a powerful technique for dependent type theories. To promote the use of machine-assisted reasoning, our development is accompanied by a short mechanized proof script, of less than 1000 lines of code, developed using the Coq proof assistant Coq Development Team (2019).

We have streamlined our proof through a number of means: the careful selection of the features that we include in the object type system and the results that we prove about it, in addition to the judicious use of automation. Our language is small, but includes enough to be illustrative. For example, we eschew inductive datatypes or W-types, but we do include propositional equality and booleans to capture the challenges presented by indexed types and dependent pattern matching. We do not show the decidability of type checking, nor do we develop a PER semantics, but we prove logical consistency, which states that empty types are not inhabited in an empty context, and demonstrate how our consistency proof can be extended (at a moderate cost of 500 lines of code) to show the existence of $\beta\eta$-normal forms for well-typed open *and* closed terms. We include a full predicative universe hierarchy and type-level computation to demonstrate the logical strength of the approach.

More concretely, our paper makes the following contributions.

- In Section 2, we introduce λ^{Π}, the dependent type theory of interest. A key design choice that impacts our proofs is the use of an untyped conversion rule, inspired by Pure Type Systems (Barendregt, 1991), and specified through parallel reduction (Takahashi, 1995; Barendregt, 1993).
- In Section 3, we formulate logical consistency for λ^{Π}, the property of interest, to motivate a logical relation. We define the relation first inductively and then later prove that it is a partial function. Based on this definition, we define semantic typing and prove the fundamental theorem, from which consistency follows as a corollary (Section 4). Thanks to the design of λ^{Π}, our proof showcases the special treatment required to model many of the most common features of dependent type theories, thus making our proof applicable to a broad range of type systems.

- We strengthen our logical relation to prove the existence of β-normal forms (Section 5) and $\beta\eta$-normal forms (Section 6) for well-typed open terms. The modifications made to our initial logical relation are small and closely mirror the necessary extensions for a simply-typed language. We use this part to show that once we have established the base techniques, we can port ideas from proofs about simpler languages to the dependently typed setting.
- We mechanize all our proofs using the Coq proof assistant, with 957 lines of code for the consistency proof and a moderate increase to 1568 lines of code for the normalization proof. We discuss the specifics related to our choice of Coq as our metatheory, including our use of off-the-shelf semantic engineering infrastructure and automation tools, in Section 7. Our proof scripts, with comments, are available to reviewers as supplementary materials.
- We compare our work to existing proofs by logical relations and other proof techniques for proving consistency and normalization. We provide an overview of this prior work (Section 8) and also give an in-depth explanation on how various design decisions affect the size of our proof and its extensibility to additional features (Section 9).

The result of our work is an artifact that an interested researcher can navigate and understand. We accompany this short mechanized proof with an informal description, presented here using set-theoretic notation and terminology so that it is accessible to readers with a general mathematical background. That said, our explanations do not stray too far away from our proof scripts. We link each lemma directly to its counterpart in the proof script, anticipating that readers may wish to see how these results may be expressed and verified in a proof assistant. The typeset proofs purposefully follow the mechanized proofs closely while avoiding, as much as possible, artifacts specific to Coq.

Not only does this close connection aid readers that wish to, like us, adopt proof assistants for their day-to-day use, but we also find that this precision is important for conveying the proof technique itself. Unlike properties that are derivable through syntactic means, proofs by logical relations make demands on the strength of the metalogic in which they are expressed. An informal proof that attempts to be agnostic or ambiguous about the underlying metatheory requires substantial effort from the reader to understand whether it is definable in a given ambient logic.

2 Specification of a Dependent Type Theory

Terms

a, b, c, p, A, B	$::=$	$\mathbf{Set}_i \mid x \mid \mathbf{Void}$	universes, variables, empty type
	$\mid$	$\Pi x{:}A.B \mid \lambda x.a \mid a\,b$	function types, abstractions, applications
	$\mid$	$a \sim b \in A \mid \mathbf{refl} \mid \mathbf{J}\ c\ a\ b\ p$	equality types, reflexivity proof, J eliminator
	$\mid$	$\mathbf{Bool} \mid \mathbf{true} \mid \mathbf{false}$	boolean type, true, false
	$\mid$	$\mathbf{if}\ a\ \mathbf{then}\ b_0\ \mathbf{else}\ b_1$	conditional expression

Substitutions	*Typing Contexts*
$\rho \in \mathit{Var} \to \mathit{Term}$	$\Gamma ::= \cdot \mid \Gamma, x : A$

Fig. 1. Syntax of λ^Π

In this section, we present the dependent type theory, λ^Π, whose logical consistency will be proven in Section 4.

The syntax of λ^Π can be found in Figure 1. As a dependent type theory, terms and types are collapsed into the same syntactic category. The type $\mathbf{Set}_i$ represent a universe type, annotated by its universe level, a natural number i. Abstractions $\lambda x.a$ and dependent function types $\Pi x{:}A.B$ are binding forms for the variable x in the body of the function and codomain of the function type. [1] We use the notation $A \to B$ when the output type B is not dependent on the input variable. For simplicity, we omit the type annotations in the abstraction forms. We discuss how the inclusion of type annotations can affect our development in Section 6, where we extend our consistency result to the existence of $\beta\eta$-normal forms. We include in λ^Π the intensional identity type $a \sim b \in A$ whose proofs can be eliminated by the J-eliminator $\mathbf{J}\ c\ a\ b\ p$, where p is an equality proof between a and b, and c is the term whose type is to be casted. Finally, λ^Π includes booleans, with standard syntax.

Our reduction and typing relations are defined in terms of *simultaneous substitutions*, ρ, which are mappings from variables to terms. We use **id** as the identity substitution. The extension operation, $(\rho[x \mapsto a])$, updates the substitution ρ to map the variable x to a rather than $\rho(x)$.

The substitution operator, which takes the form $a\{\rho\}$, traverses the syntax of a and replaces each variable x with the term $\rho(x)$. When traversing under binders (e.g. in the $(\lambda x.a)\{\rho\}$ case), it must be the case that ρ maps the bound variable to itself and that the bound variable does not appear freely in the application of the substitution to any other variable.

[1] In the exposition in this paper, binding forms are equal up to α-conversion and we adopt the Barendregt Variable Convention Barendregt (1985), which lets us assume that bound variables are distinct. In some places, we are informal about the treatment of variables and substitution; our mechanized proofs make these notions precise by using de Bruijn indices de Bruijn (1994).

The substitution operator is referred to as simultaneous substitution as it substitutes for all variables at once. It is possible to recover single substitution by composing the extension operator and the identity substitution: $a\{b/x\} := a\{\mathbf{id}[x \mapsto b]\}$.

When reasoning about logical relations, we find it more convenient to formulate simultaneous substitution directly rather than recovering it from single substitution. In particular, this shows up in the definition of semantic typing in Section 4, which relies on simultaneous substitution.

2.1 Definitional equality via parallel reduction

$\boxed{a \Rightarrow b}$ *(Parallel Reduction)*

$$\text{P-AppAbs}\quad \frac{a_0 \Rightarrow a_1 \qquad b_0 \Rightarrow b_1}{(\lambda x.a_0)\ b_0 \Rightarrow a_1\{b_1/x\}} \qquad\qquad \text{P-IfTrue}\quad \frac{b_0 \Rightarrow b_1}{\textbf{if true then}\ b_0\ \textbf{else}\ c_0 \Rightarrow b_1}$$

$$\text{P-IfFalse}\quad \frac{c_0 \Rightarrow c_1}{\textbf{if false then}\ b_0\ \textbf{else}\ c_0 \Rightarrow c_1} \qquad\qquad \text{P-JRefl}\quad \frac{c_0 \Rightarrow c_1}{\mathbf{J}\ c_0\ a_0\ b_0\ \textbf{refl} \Rightarrow c_1}$$

Fig. 2. Parallel reduction (β-rules only)

Before we specify the typing rules, we first specify the equational theory used in the conversion rule (rule T-Conv in Figure 3). The equivalence relation used in this rule is often referred to *definitional equality* in dependent type theories because it defines the equivalence that the syntactic type system works up to.

In λ^{Π}, we use a relation called *convertibility* for definitional equality. Two terms are convertible, if they reduce to a common form. The reduction that we use is called *parallel reduction*, written $a \Rightarrow b$. The notation $a \Rightarrow^* b$ indicates the transitive closure of parallel reduction.

Definition 2.1 (Convertibility)**.** Two terms a_0 and a_1 are *convertible*, written $a_0 \Leftrightarrow a_1$, if there exists some term b such that $a_0 \Rightarrow^* b$ and $a_1 \Rightarrow^* b$.

The definition of the parallel reduction relation, appears in Figure 2. (For brevity, the reflexivity and congruence rules of this relation are omitted from this figure).

We prove, through standard techniques Takahashi (1995); Wadler et al. (2022), the following properties of parallel reduction.

Lemma 2.2 (Par Refl[2])**.** *For all terms a, $a \Rightarrow a$.*

Lemma 2.3 (Par cong[3])**.** *If $a_0 \Rightarrow a_1$ and $b_0 \Rightarrow b_1$, then $a_0\{b_0/x\} \Rightarrow a_1\{b_1/x\}$.*

[2] `join.v:Par_refl` [3] `join.v:par_cong`

Corollary 2.4 (Par subst[4])**.** *If $a_0 \Rightarrow a_1$, then $a_0\{b/x\} \Rightarrow a_1\{b/x\}$ for arbitrary b.*

Lemma 2.5 (Par diamond[5])**.** *If $a \Rightarrow b_0$ and $a \Rightarrow b_1$, then there exists some term c such that $b_0 \Rightarrow c$ and $b_1 \Rightarrow c$.*

Convertibility is an equivalence relation. The key step in proving transitivity is showing the diamond property for parallel reduction.

Lemma 2.6 (Convertibility refl[6])**.** *For all terms a, $a \Leftrightarrow a$.*

Lemma 2.7 (Convertibility sym[7])**.** *If $a \Leftrightarrow b$, then $b \Leftrightarrow a$.*

Lemma 2.8 (Convertibility trans[8])**.** *If $a_0 \Leftrightarrow a_1$ and $a_1 \Leftrightarrow a_2$, then $a_0 \Leftrightarrow a_2$.*

The convertibility relation that we use for conversion in λ^{Π} is unusual in that it is directly defined via parallel reduction, instead of using the related notion of β-equivalence (Barendregt, 1991; Coquand and Paulin, 1990). This choice does not change the language definition; a detailed argument of the equivalence between $a \Leftrightarrow b$ and untyped β-equivalence can be found in Barendregt (1993) and Takahashi (1995). However, this choice simplifies later proofs, as we discuss in Section 9.

Our definition of equality is untyped: the judgement does not require the two terms to type check and have the same type. The use of an untyped relation for conversion is similar to Barendregt's Pure Type Systems Barendregt (1991) and differs from MLTT (Martin-Löf, 1975), where the judgmental equality takes the form $\Gamma \vdash a \equiv b : A$. By working with an untyped judgement, we can establish its properties independently from the type system and the logical relation, using well-established syntactic approaches. Siles and Herbelin (2012) show the equivalence of Barendregt's Pure Type System, which employs untyped equality, and its variant that uses typed judgmental equality. This assures us that we do not lose generality working with a system with untyped conversion. We compare this definition with type-directed approaches to equality in Section 9.

[4] `join.v:par_subst`

[5] `join.v:par_confluent`

[6] `join.v:Coherent_reflexive`

[7] `join.v:Coherent_symmetric`

[8] `join.v:Coherent_transitive`

2.2 Syntactic Typing

$\boxed{\vdash \Gamma}$ *(Context Well-Formedness)*

$$\textsc{Ctx-Empty}\quad \frac{}{\vdash \cdot} \qquad\qquad \textsc{Ctx-Cons}\quad \frac{\vdash \Gamma \qquad \Gamma \vdash A : \mathbf{Set}_i \qquad x \notin dom(\Gamma)}{\vdash \Gamma, x : A}$$

$\boxed{\Gamma \vdash a : A}$ *(Typing)*

$$\textsc{T-Var}\quad \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{T-Set}\quad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{Set}_i : \mathbf{Set}_{(1+i)}} \qquad \textsc{T-Pi}\quad \frac{\Gamma \vdash A : \mathbf{Set}_i \qquad \Gamma, x : A \vdash B : \mathbf{Set}_i}{\Gamma \vdash \Pi x{:}A.B : \mathbf{Set}_i}$$

$$\textsc{T-Abs}\quad \frac{\Gamma \vdash \Pi x{:}A.B : \mathbf{Set}_i \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : \Pi x{:}A.B} \qquad \textsc{T-App}\quad \frac{\Gamma \vdash b : \Pi x{:}A.B \qquad \Gamma \vdash a : A}{\Gamma \vdash b\ a : B\{a/x\}} \qquad \textsc{T-Conv}\quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash B : \mathbf{Set}_i \qquad A \Leftrightarrow B}{\Gamma \vdash a : B}$$

$$\textsc{T-Void}\quad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{Void} : \mathbf{Set}_i} \qquad \textsc{T-Bool}\quad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{Bool} : \mathbf{Set}_i} \qquad \textsc{T-True}\quad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{true} : \mathbf{Bool}} \qquad \textsc{T-False}\quad \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{false} : \mathbf{Bool}}$$

$$\textsc{T-If}\quad \frac{\Gamma, x : \mathbf{Bool} \vdash A : \mathbf{Set}_i \qquad \Gamma \vdash a : \mathbf{Bool} \qquad \Gamma \vdash b_0 : A\{\mathbf{true}/x\} \qquad \Gamma \vdash b_1 : A\{\mathbf{false}/x\}}{\Gamma \vdash \mathbf{if}\ a\ \mathbf{then}\ b_0\ \mathbf{else}\ b_1 : A\{a/x\}}$$

$$\textsc{T-Eq}\quad \frac{\Gamma \vdash A : \mathbf{Set}_j \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a \sim b \in A : \mathbf{Set}_i} \qquad \textsc{T-Refl}\quad \frac{\vdash \Gamma \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathbf{refl} : a \sim a \in A}$$

$$\textsc{T-J}\quad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash A : \mathbf{Set}_j \qquad \Gamma \vdash p : a \sim b \in A \qquad \Gamma, x : A, y : x \sim a \in A \vdash B : \mathbf{Set}_i \qquad \Gamma \vdash c : B\{a, \mathbf{refl}/x, y\}}{\Gamma \vdash \mathbf{J}\ c\ a\ b\ p : B\{b, p/x, y\}}$$

Fig. 3. Syntactic typing for λ^{Π}

Figure 3 gives the full typing rules for λ^{Π}. The premises wrapped in gray boxes can be shown to be admissible syntactically, though some of them are required to strengthen the inductive hypothesis of the fundamental theorem.

The typing rules of λ^{Π} are standard for dependent type theories. The variable rule, rule T-Var, uses the auxiliary relation $x : A \in \Gamma$, that holds when a variable declaration is found in the typing context. The typing of universes ensures that each one belongs to the next higher level. Rule T-Pi ensures predicative quantification by requiring that all parts of the type be typeable at the same universe level. Rule T-Abs ensures that all functions have well-formed dependent types. In an application (rule T-App) the argument is substituted for the variable in the result type.

Rule T-Conv uses the convertibility relation from earlier as our equality judgment for type conversion.

The elimination form for booleans, rule T-If, demonstrates dependent pattern matching. The result type of this expression, $A\{a/x\}$, is composed of some *motive* A, a type where its single free variable has been replaced with the condition of the if expression. When typing the true branch, this substitution replaces the variable by **true**, and similarly for the false branch. As a result, the type system communicates the information gained from the test to each of the branches of the expression.

A similar sort of dependent pattern matching occurs when eliminating identity types. Such types are checked for well-formedness with rule T-Eq and introduced by rule T-Refl. In rule T-J, the elimination form, the subterm p is a proof of an equality between a and b. The subterm c is the body of the elimination form. In this rule, B is the motive and has two free variables. When checking c, the substitution for these variables changes from b to a and from p to **refl**, witnessing the information gained through dependent pattern matching.

The universe hierarchy and the boolean base type gives λ^{Π} the ability to compute a type using a term as input, a feature commonly referred to as large elimination. For example, we can type check the function $\lambda x.\mathbf{if}\ x\ \mathbf{then\ Bool\ else\ Void}$, which returns either **Bool** or **Void** depending on whether its input is **true** or **false**.

3 Logical Relation

Before we define our logical relation, we first formally specify the consistency property that we want to prove.

Theorem 3.1 (Logical Consistency)**.** *The judgment* $\cdot \vdash a : \mathbf{Void}$ *is not derivable.*

The property can be formulated in a simply typed language, where **Void** is similarly defined as a type that has no term. A related property, referred to as the termination property (for closed terms), is commonly used in introductory materials such as Skorstengaard (2019), Pierce (2002), and Harper (2022*a*) to motivate the need for a logical relation.

A naive attempt to proving Theorem 3.1 by induction on the derivation $\cdot \vdash a : \mathbf{Void}$ would succeed at almost all cases except for rule T-App. In the application case, we are given $\cdot \vdash b : \Pi x{:}A.B$ and $\cdot \vdash a : A$, and the equality that $B\{a/x\} = \mathbf{Void}$. Our goal is to show that $\cdot \vdash b\ a : \mathbf{Void}$ is not possible. However, note that there is nothing we know of b or a from the induction hypothesis because neither $\Pi x{:}A.B$ nor A is equal to **Void**. We have no way of deriving a contradiction from $\cdot \vdash b\, a : \mathbf{Void}$. The takeaway from this failed attempt is that, in order to derive the consistency, we need to know something about types other than **Void**. From a pragmatic point of view, proof by logical relation can be seen as a sophisticated way of strengthening the induction hypothesis. From the strengthened property, the fundamental theorem, we will be able to derive consistency as a corollary.

The complexity of applying proof by logical relation to dependent types stems from the fact that the logical relation is much harder to define. In simply typed languages, the logical relation is defined as a recursive function over the type A. In dependent types, the type A can take the form $(\lambda x.x)\ \mathbf{Bool}$. To assign meaning to this type, we need to first reduce it to **Bool**. However, we cannot write a function that performs the reduction because we do not know the termination of well-typed terms a priori. As a result, we define the logical relation as an inductively defined relation, reminiscent of how we specify the reduction graph of a partial function; the functionality of the relation can later be recovered in Lemma 3.7.

3.1 Definition of the Logical Relation

$\boxed{[\![A]\!]^i_I \searrow S}$ *(Logical Relation)*

$$\textsc{I-Set}\ \frac{j < i}{[\![\mathbf{Set}_j]\!]^i_I \searrow I(j)} \qquad \textsc{I-Void}\ \frac{}{[\![\mathbf{Void}]\!]^i_I \searrow \emptyset} \qquad \textsc{I-Bool}\ \frac{}{[\![\mathbf{Bool}]\!]^i_I \searrow \{a \mid a \Rightarrow^* \mathbf{true} \vee a \Rightarrow^* \mathbf{false}\}}$$

$$\textsc{I-Eq}\ \frac{}{[\![a \sim b \in A]\!]^i_I \searrow \{p \mid p \Rightarrow^* \mathbf{refl} \wedge a \Leftrightarrow b\}} \qquad \textsc{I-Red}\ \frac{A \Rightarrow B \qquad [\![B]\!]^i_I \searrow S}{[\![A]\!]^i_I \searrow S}$$

$$\textsc{I-Pi}\ \frac{[\![A]\!]^i_I \searrow S \qquad F \in S \to \mathcal{P}(\mathit{Term}) \qquad \forall a, \text{if } a \in S, \text{then } [\![B\{a/x\}]\!]^i_I \searrow F(a)}{[\![\Pi x{:}A.B]\!]^i_I \searrow \{b \mid \forall a, \text{if } a \in S, \text{then } b\, a \in F(a)\}}$$

Fig. 4. Logical relation for λ^{Π}

The logical relation for λ^{Π}, which takes the form $[\![A]\!]^i_I \searrow S$, is defined as an inductively generated relation (Figure 4). Metavariables A and i stand for terms and natural numbers respectively, as introduced earlier in Figure 1. The metavariables I and S are sets with the following signatures.

$$I \in \{j \mid j < i\} \to \mathcal{P}(\mathit{Term}) \qquad\qquad S \in \mathcal{P}(\mathit{Term})$$

The notation $\mathcal{P}(\mathit{Term})$ denotes the powerset of the set of λ^{Π} terms. The function I is a family of sets of terms indexed by natural numbers strictly less than the parameter i, which represents the current universe level. In rule I-Set, the function I is used to define the meaning of universes that are strictly smaller than the current level i. The restriction $j < i$ in rule I-Set ensures the predicativity of the system.

We tie the knot and obtain an interpretation for all universe levels below. The judgment $\llbracket A \rrbracket^i \searrow S$ reads that the type A is a level-i type *semantically* inhabited by terms from the set S.

Definition 3.2 (Logical relation for all universe levels)**.** Define $\llbracket A \rrbracket^i \searrow S$ recursively through the well-foundedness of the $<$ relation on natural numbers.

$$\llbracket A \rrbracket^i \searrow S := \llbracket A \rrbracket^i_I \searrow S, \text{ where } I(j) := \{A \mid \exists S, \llbracket A \rrbracket^j \searrow S\} \text{ for } j < i$$

Definition 3.2 explains how the $j < i$ constraint in rule I-Set makes our system predicative; the interpretation of the i_{th} universe is only dependent on universes strictly lower than i, which have already been defined. This restriction ensures that that the relation is well-defined: without it the definition of $\llbracket A \rrbracket^i \searrow S$ would not be well-founded; $\llbracket A \rrbracket^i_I \searrow S$ would call I on universe levels greater than or equal to i, which are yet to be defined.

By unfolding Definition 3.2, we can show that the same introduction rules for $\llbracket A \rrbracket^i_I \searrow S$ are admissible for $\llbracket A \rrbracket^i \searrow S$. For example, we can prove the following derived rules:

$$\textsc{IR-Void}\ \frac{}{\llbracket \mathbf{Void} \rrbracket^i \searrow \emptyset} \qquad\qquad \textsc{IR-Set}\ \frac{j < i}{\llbracket \mathbf{Set}_j \rrbracket^i \searrow \{A \mid \exists S, \llbracket A \rrbracket^j \searrow S\}}$$

In most informal presentations, instead of defining the logical relation in two steps as we have shown above, the rules for $\llbracket A \rrbracket^i \searrow S$ are given directly, with the implicit understanding that the relation is an inductive definition nested inside a recursive function over the universe level i. We choose the more explicit definition not only because it is directly definable in proof assistants that lack induction-recursion, but also because it makes clear the induction principle we are allowed to use when reasoning about $\llbracket A \rrbracket^i \searrow S$.

We next take a closer look at the inductive relation $\llbracket A \rrbracket^i_I \searrow S$, defined in Figure 4. Rules I-Void and I-Bool capture terms that *behave like* the inhabitants of the **Void** and **Bool** types under an empty context. For example, the **Void** type should not have any inhabitants under the empty context, where as the **Bool** type only contains terms that reduce to **true** or **false**. Note that the characterization of **Bool** (and other inhabited types) in our logical relation does not always correspond to well-typed or even closed terms. For example, the term **if false then Void true else true** is ill-typed under the empty context but still belongs to the set $\{a \mid a \Rightarrow^* \mathbf{true} \lor a \Rightarrow^* \mathbf{false}\}$ since it evaluates to **true**. The independence of syntactic typing in our logical relation allows our semantic typing definition in Section 4 to be meaningful on its own. Furthermore, not having to embed scoping information into the logical relation avoids extra bookkeeping and the need for a Kripke-style logical relation when we extend our logical relation to prove the existence of β-normal forms (Section 5).

Rule I-Eq says that an equality type $a \sim b \in A$ corresponds to the set of terms that reduce to **refl** when $a \Leftrightarrow b$ also holds and otherwise corresponds to the empty set. Conditions like $a \Leftrightarrow$

b are typically required for indexed types, of which equality types are an instance. Rule I-Red enables us to reduce types in order to assign meanings. Recall the type expression $(\lambda x.x)\ \mathbf{Bool}$. Rule I-Red says that to know that $[\![(\lambda x.x)\ \mathbf{Bool}]\!]^i_I \searrow S$ for some S, it suffices to show that $[\![\mathbf{Bool}]\!]^i_I \searrow S$ since $(\lambda x.x)\ \mathbf{Bool} \Rightarrow \mathbf{Bool}$. The derivation that $[\![(\lambda x.x)\ \mathbf{Bool}]\!]^i_I \searrow \{a \mid a \Rightarrow^* \mathbf{true} \vee a \Rightarrow^* \mathbf{false}\}$ therefore follows by composing rule I-Red and rule I-Bool.

Rule I-Pi is the most complex rule in our logical relation. Instead of explaining it directly, we first consider the following simplified version, rule I-PiAlt, that follows directly from rule I-Pi.

I-PiAlt

$$\frac{[\![A]\!]^i_I \searrow S \qquad \forall a, \text{if } a \in S, \text{then } \exists S_0, [\![B\{a/x\}]\!]^i_I \searrow S_0}{[\![\Pi x{:}A.B]\!]^i_I \searrow \{b \mid \forall a, \text{if } a \in S, \text{then } \forall S_0, \text{if } [\![B\{a/x\}]\!]^i_I \searrow S_0, \text{then } b\ a \in S_0\}}$$

Rule I-PiAlt directly captures the meaning of a well-behaved dependent function type. The precondition of the rule says that the function type $\Pi x{:}A.B$ has an interpretation if its input type A can be interpreted as some set S, and for all terms $a \in S$, the type $B\{a/x\}$, obtained by substituting a into the output type B, has some semantic interpretation S_0. In its conclusion, the interpretation of $\Pi x{:}A.B$ is the set of terms b, such that for all $a \in S$, where S is an interpretation of A, the application form $b\ a$ belongs to all possible interpretations of $B\{a/x\}$ (the pre-condition ensures at least one interpretation exists for each $B\{a/x\}$ where $a \in S$).

Lemma 3.3 (I-PiAlt derivability)**.** *Rule* I-PiAlt *is derivable from rule* I-Pi*.*

Proof The precondition $\forall a, \text{if } a \in S, \text{then } \exists S_0, [\![B\{a/x\}]\!]^i_I \searrow S_0$ from rule I-PiAlt immediately induces a function $F \in S \to \mathcal{P}(\mathit{Term})$ such that $\forall a, \text{if } a \in S, \text{then } [\![B\{a/x\}]\!]^i_I \searrow F(a)$, which is exactly what we need to apply rule I-Pi. ■

In fact, while rule I-PiAlt is an instantiation of rule I-Pi, these two rules are equivalent in the sense that every derivation involving rule I-Pi can be systematically replaced by rule I-PiAlt. This equivalence follows directly from the fact that the logical relation is a partial function, a result we will show in Lemma 3.7. The preconditions of rule I-Pi, when combined with the functionality of the logical relation, uniquely determine the function $F \in S \to \mathcal{P}(\mathit{Term})$ to be the functional relation $\{(a, S_0) \mid \text{if } a \in S, \text{then } [\![B\{a/x\}]\!]^i_I \searrow S_0\}$. This result is formally shown through the improved inversion lemma for function types (Lemma 3.8).

Unfortunately, we cannot define the function case of our logical relation directly using rule I-PiAlt since the occurrence of $[\![B\{a/x\}]\!]^i_I \searrow S_0$ in its conclusion not only violates the syntactic strict positivity constraint required in proof assistants, but is genuinely non-monotone when we treat the inductive definition as the fixed point of an endofunction over the domain of relations. Intuitively, the failure of monotonicity stems from the fact that the witness picked in the precondition is not necessarily the same witness being referred to in the post condition as the relation grows, whereas the function F in rule I-Pi ‘‘fixes’’ the witnesses S_0 as $F(a)$ for each $a \in S$, thus preventing the set of witnesses from growing.

While it might be possible to restrict the domain with additional constraints such as functionality and inversion properties to justify the well-definedness of our inductive relation with rule I-PiAlt, we opt for our current rule I-Pi that immediately produces a well-defined inductive relation and usable induction principle. The slight disadvantage of rule I-Pi is that we need to construct the function F each time we apply it, though this is mitigated by the derivability of rule I-PiAlt and the alternative Π inversion principle (Lemma 3.8).

3.2 Properties about the Logical Relation

In the rest of this section, we develop the theory of our logical relation with the goal of showing four key facts: irrelevance (Lemma 3.6), functionality (Lemma 3.7), cumulativity (Lemma 3.9), and the backward closure property (Lemma 3.12). For the majority of the properties that we prove in this section, we do not need any information about the parameterized function I. Each property about $\llbracket A \rrbracket^i \searrow S$ follows as a corollary of a property about $\llbracket A \rrbracket^i_I \searrow S$ with no or few assumptions imposed on I. As a result, we usually state our lemmas in terms of $\llbracket A \rrbracket^i_I \searrow S$ without duplicating them in terms of $\llbracket A \rrbracket^i \searrow S$.

First, we prove a family of simple properties, which we refer to as inversion principles for our logical relation. Given $\llbracket A \rrbracket^i_I \searrow S$ where A is in some head form such as **Bool** or $\Pi x{:}A_0.B_0$, the inversion lemma allows us to say something about the set S. Its proof is simple, but we sketch out the case for functions to help readers confirm their understanding of rule I-Pi.

Lemma 3.4 (Inversion of the logical relation)**.**

1. [9] *If* $\llbracket \mathbf{Void} \rrbracket^i_I \searrow S$*, then* $S = \emptyset$.
2. [10] *If* $\llbracket \mathbf{Bool} \rrbracket^i_I \searrow S$*, then* $S = \{a \mid a \Rightarrow^* \mathbf{true} \lor a \Rightarrow^* \mathbf{false}\}$.
3. [11] *If* $\llbracket a \sim b \in A \rrbracket^i_I \searrow S$*, then* $S = \{p \mid p \Rightarrow^* \mathbf{refl} \land a \Leftrightarrow b\}$.
4. [12] *If* $\llbracket \Pi x{:}A.B \rrbracket^i_I \searrow S_1$*, then there exists* S, F *such that:*
   - $\llbracket A \rrbracket^i_I \searrow S$
   - $F \in S \to \mathcal{P}(Term)$
   - $\forall a$*, if* $a \in S$*, then* $\llbracket B\{a/x\} \rrbracket^i_I \searrow F(a)$
   - $S_1 = \{b \mid \forall a, \text{if } a \in S, \text{then } b\ a \in F(a)\}$
5. [13] *If* $\llbracket \mathbf{Set}_j \rrbracket^i_I \searrow S$*, then* $j < i$ *and* $S = I(j)$.

Proof As mentioned earlier, we only show the inversion property for the function type. We start by inducting over the derivation of $\llbracket \Pi x{:}A.B \rrbracket^i_I \searrow S$. There are only two possible cases we need to consider.

Rule I-Pi: Immediate.
Rule I-Red: We are given that $\llbracket \Pi x{:}A.B \rrbracket^i_I \searrow S_1$. We know that there exists some A_0 and B_0 such that $\Pi x{:}A.B \Rightarrow \Pi x{:}A_0.B_0$ and $\llbracket \Pi x{:}A_0.B_0 \rrbracket^i_I \searrow S_1$. From the induction hypothesis, there exists S and F such that :

[9] `semtyping.v:InterpExt_Void_inv`

[10] `semtyping.v:InterpExt_Bool_inv`

[11] `semtyping.v:InterpExt_Eq_inv`

[12] `semtyping.v:InterpExt_Fun_inv`

[13] `semtyping.v:InterpExt_Univ_inv`

- $[\![A_0]\!]^i_I \searrow S$
- $F \in S \to \mathcal{P}(Term)$
- $\forall a$, if $a \in S$, then $[\![B_0\{a/x\}]\!]^i_I \searrow F(a)$
- $S_1 = \{b \mid \forall a, \text{if } a \in S, \text{then } b\,a \in F(a)\}$

By inverting the derivation of $\Pi x{:}A.B \Rightarrow \Pi x{:}A_0.B_0$, we derive $A \Rightarrow A_0$ and $B \Rightarrow B_0$. By Lemma 2.4, we have $B\{a/x\} \Rightarrow B_0\{a/x\}$ for all a. As a result, by rule I-RED, the same S and F additionally satisfy the following properties.

- $[\![A]\!]^i_I \searrow S$
- $\forall a$, if $a \in S$, then $[\![B\{a/x\}]\!]^i_I \searrow F(a)$

These properties are exactly what we need to finish the proof.

■

Rule I-RED bakes into the logical relation the backward preservation property. That is, given $[\![A]\!]^i_I \searrow S$, if $B \Rightarrow^* A$, then $[\![B]\!]^i_I \searrow S$ also holds. The following property shows that preservation holds in the usual forward direction too.

Lemma 3.5 (Forward preservation[14]). *If $[\![A]\!]^i_I \searrow S$ and $A \Rightarrow B$, then $[\![B]\!]^i_I \searrow S$.*

Proof We carry out the proof by induction over the derivation of $[\![A]\!]^i_I \searrow S$.

The only interesting case is rule I-RED. Given that $A \Rightarrow B_0$ and $[\![B_0]\!]^i_I \searrow S$, we need to show for all B_1 such that $A \Rightarrow B_1$, we have $[\![B_1]\!]^i_I \searrow S$. By the diamond property of parallel reduction (Lemma 2.5), there exists some term B such that $B_0 \Rightarrow B$ and $B_1 \Rightarrow B$. By the induction hypothesis, we deduce $[\![B]\!]^i_I \searrow S$ from $B_0 \Rightarrow B$ and $[\![B_0]\!]^i_I \searrow S$. By rule I-RED and $B_1 \Rightarrow B$, we conclude that $[\![B]\!]^i_I \searrow S$.

The remaining cases all fall from induction hypotheses and basic properties about convertibility and parallel reduction we have established in Section 2. ■

From Lemma 3.5 and rule I-RED, we can easily derive the following corollary that two convertible types can always interpret into the same set. We adopt the terminology from Adjedj et al. (2024) and refer to this property as irrelevance.

Corollary 3.6 (Irrelevance of logical relation[15]). *If $[\![A]\!]^i_I \searrow S$ and $A \Leftrightarrow B$, then $[\![B]\!]^i_I \searrow S$.*

Because the definition of our logical relation is an inductive relation, it is not immediately obvious why each type A can only uniquely correspond to one set S. The following lemma shows that our logical relation is indeed functional.

Lemma 3.7 (Logical relation is functional[16]). *If $[\![A]\!]^i_I \searrow S_0$ and $[\![A]\!]^i_I \searrow S_1$, then $S_0 = S_1$.*

Proof The proof proceeds by induction over the derivation of the first premise $[\![A]\!]^i_I \searrow S_0$. All cases that are not rule I-RED follow immediately from Lemma 3.4, the inversion properties.

[14] `semtyping.v:InterpExt_preservation`

[15] `semtyping.v:InterpUnivN_Coherent`

[16] `semtyping.v:InterpExt_deterministic`

For rule I-Red, we are given that there exists some B such that $A \Rightarrow B$ and $[\![B]\!]^i_I \searrow S_0$. Our goal is to show that given $[\![A]\!]^i_I \searrow S_1$ for some S_1, we have $S_0 = S_1$. By the preservation property (Lemma 3.5), we know that $[\![B]\!]^i_I \searrow S_1$ since $A \Rightarrow B$. The statement $S_0 = S_1$ then immediately follows from the induction hypothesis. ■

Lemma 3.7 enables us to show the following improved inversion lemma for function types whose statement is free of the relation F, analogous to the derivable rule I-PiAlt.

Lemma 3.8 (Pi Inversion Alt[17])**.** *Suppose $[\![\Pi x{:}A.B]\!]^i_I \searrow S$, then there exists some S_0 such that the following constraints hold:*

- $[\![A]\!]^i_I \searrow S_0$
- $\forall a$, *if* $a \in S_0$, *then* $\exists S_1$, $[\![B\{a/x\}]\!]^i_I \searrow S_1$
- $S = \{b \mid \forall a$, *if* $a \in S_0$, *then* $\forall S_1$, *if* $[\![B\{a/x\}]\!]^i_I \searrow S_1$, *then* $b\ a \in S_1\}$

Proof Immediate from Lemmas 3.4 and 3.7. ■

The next lemma shows that our logical relation satisfies cumulativity. That is, if a type has an interpretation at a lower universe level, then we can obtain the same interpretation at a higher universe level.

Lemma 3.9 (Logical relation cumulativity[18])**.** *If $[\![A]\!]^{i_0}_I \searrow S$ and $i_0 < i_1$, then $[\![A]\!]^{i_1}_I \searrow S$.*

Proof Trivial by structural induction over the derivation of $[\![A]\!]^{i_0}_I \searrow S$. ■

Note that in the statement of Lemma 3.9, we implicitly assume that I is defined on the set of natural numbers less than i_1.

Corollary 3.10 (Logical relation is functional with different levels[19])**.** *If $[\![A]\!]^{i_0}_I \searrow S_0$ and $[\![A]\!]^{i_1}_I \searrow S_1$, then $S_0 = S_1$.*

Proof Immediate from Lemmas 3.7 and 3.9. ■

Definition 3.11 (Sets closed under expansion)**.** We say that a set of terms S is closed under expansion if given $a \in S$, then $b \in S$ for all $b \Rightarrow a$.

The final property we want to show is that the output set S from the logical relation is closed under expansion. Unlike the previous lemmas, we directly state the lemma in terms of $[\![A]\!]^i \searrow S$ rather than $[\![A]\!]^i_I \searrow S$ because we need to know something about I for this property to hold in the rule I-Set case.

Lemma 3.12 (Interpreted sets are closed under expansion[20])**.** *If $[\![A]\!]^i \searrow S$, then the set S is closed under expansion.*

Proof By the definition of $[\![A]\!]^i \searrow S$, we unfold $[\![A]\!]^i \searrow S$ by one step into $[\![A]\!]^i_I \searrow S$ where $I(j) := \{A \mid \exists S, [\![A]\!]^j \searrow S\}$. We then proceed by induction over the derivation of $[\![A]\!]^i_I \searrow S$.

[17] `semtyping.v:InterpExt_Fun_inv_nopf`

[18] `semtyping.v:InterpExt_cumulative`

[19] `semtyping.v:InterpExt_deterministic'`

[20] `semtyping.v:InterpUnivN_back_clos`

All cases are trivial except for the rule I-Set case, where we want to show that the set $I(j)$ is closed under expansion for all $j < i$. However, by the definition of I, we know that $A \in I(j)$ if and only if there exists some S such that $[\![A]\!]^j_I \searrow S$. By rule I-Red, we must also have $B \in I(j)$ for all $B \Rightarrow A$. ■

4 Semantic Typing and Consistency

In this section, we show that all closed, well-typed terms are contained within their type-indexed sets. In other words, $\cdot \vdash a : A$ implies $[\![A]\!]^i \searrow S$ and $a \in S$. This result gives us consistency because we know that $[\![\textbf{Void}]\!]^i \searrow S$ is defined, and that S must be the empty set. Therefore, if there were some closed, well-typed term of type **Void**, it would need to be a member of the empty set, a contradiction.

To prove this result, we define a notion of semantic typing based on the logical relation we have defined in Section 3 and prove the fundamental lemma, which states that syntactic typing implies semantic typing. Semantic typing extends our logical relation from being a (type-indexed) family of predicates on closed terms, to a type-indexed family of predicates on open terms.

The necessity of semantic typing as an extra layer of definition on top of the logic relation can be understood in simply typed languages (Skorstengaard, 2019; Harper, 2022*a*; Pierce, 2002). In our setting, attempting to show that $\cdot \vdash a : A$ implies $[\![A]\!]^i \searrow S$ and $a \in S$ through induction over the derivation of $\cdot \vdash a : A$ will fail in rule T-Abs, where the induction hypothesis is not helpful since the body of the lambda term is typed under a non-empty context. Through the definition of semantic typing, we can state a strengthened property that is actually provable.

Definition 4.1 (Semantic well-formed substitution[21]). Define $\rho \vDash \Gamma$ when

$$\forall x, A, i, \text{ and } S, \text{ if } x : A \in \Gamma \text{ and } [\![A\{\rho\}]\!]^i \searrow S, \text{ then } \rho(x) \in S$$

The $\rho \vDash \Gamma$ notation denotes the semantic well-formedness of a substitution ρ with respect to a context Γ. For every variable x with its associated type A in the context, $\rho(x)$ is a term that inhabits all possible interpretations of the type $A\{\rho\}$. The $\forall$ quantifier in its definition might look excessive since we know from Lemma 3.7 that each type can have at most one interpretation. However, since $\rho \vDash \Gamma$ mostly appears in the position of a hypothesis, the $\forall$ statement is easy to instantiate and makes our proofs slightly easier. The few cases where we need to prove $\rho \vDash \Gamma$ are handled by the following two structural properties, the second of which depends on Lemma 3.7.

Lemma 4.2 (Well-formed ρ empty[22]). *$\rho \vDash \Gamma$ whenever Γ is the empty context.*

Lemma 4.3 (Well-formed ρ cons[23]). *If $[\![A]\!]^i \searrow S$, $a \in S$, and $\rho \vDash \Gamma$, then $\rho[x \mapsto a] \vDash \Gamma, x : A$.*

We next define semantic well-typedness.

[21] `soundness.v`:`ρ_ok` [22] `soundness.v`:`ρ_ok_nil` [23] `soundness.v`:`ρ_ok_cons`

Definition 4.4 (Semantic typing[24]). Define $\Gamma \vDash a : A$ when

$$\forall \rho,\ \text{if } \rho \vDash \Gamma \text{ then there exists some } j \text{ and } S \text{ such that } [\![A\{\rho\}]\!]^j \searrow S \text{ and } a\{\rho\} \in S$$

This definition says the term a can be semantically typed A under the context Γ if for all substitutions ρ such that $\rho \vDash \Gamma$, the type $A\{\rho\}$ can be interpreted as the set S, and $a\{\rho\} \in S$. Our definition of semantic well-typedness is standard, though dependent types add a small twist that we apply the ρ to A and require that $A\{\rho\}$ has some interpretation.

Finally, we define semantic well-formedness for contexts, analogous to the relation $\vdash \Gamma$.

Definition 4.5 (Semantic context well-formedness[25]). Define $\vDash \Gamma$ as follows.

$$\forall x : A \in \Gamma,\ \text{there exists some } i \text{ such that } \Gamma \vDash A : \mathbf{Set}_i$$

Recall that $\vdash \Gamma$ is defined inductively in terms of the syntactic typing judgment. We take a different approach here with its semantic counterpart $\vDash \Gamma$. The definition of $\vDash \Gamma$ is not telescopic: with $\vdash \Gamma$, a variable appearing earlier in the context is well-scoped under a truncated context, whereas with $\vDash \Gamma$, the types are only required to be semantically well-formed with respect to the full context, regardless of their position in Γ. Our definition of $\vDash \Gamma$ could be strengthened, though the simpler definition is sufficient for showing the fundamental lemma.

We can recover the structural rules for $\vDash \Gamma$ as lemmas.

Lemma 4.6 (Semantic context well-formedness empty[26]). *$\vDash \Gamma$ holds when Γ is empty.*

Lemma 4.7 (Semantic context well-formedness cons[27]). *If $\vDash \Gamma$ and $\Gamma \vDash A : \mathbf{Set}_i$, then $\vDash \Gamma, x : A$.*

The following lemma makes the statement $\Gamma \vDash A : \mathbf{Set}_i$ easier to work with.

Lemma 4.8 (Set Inversion[28]). *The following two statements are equivalent:*

- $\Gamma \vDash A : \mathbf{Set}_i$
- *$\forall\ \rho$, if $\rho \vDash \Gamma$, then there exists S such that $[\![(A\{\rho\})]\!]^i \searrow S$*

Proof The forward direction is immediate by Lemma 3.4. We now consider the backward direction and show that $\Gamma \vDash A : \mathbf{Set}_i$ given the second bullet.

Suppose $\rho \vDash \Gamma$, then we know that there exists some S such that $[\![(A\{\rho\})]\!]^i \searrow S$. By the definition of semantic typing, it suffices to show that there exists some j and S_0 such that $[\![\mathbf{Set}_i]\!]^j \searrow S_0$ and $A\{\rho\} \in S_0$. Pick $1 + i$ for j and $\{A \mid \exists S, [\![A]\!]^i \searrow S\}$ for S_0 and it is trivial to verify the conditions hold. ■

Next, we show some non-trivial cases of the fundamental theorem as top-level lemmas. For example, we can define the semantic analogue to the syntactic typing rule for variables (rule T-Var).

[24] soundness.v:SemWt

[25] soundness.v:SemWff

[26] soundness.v:SemWff_nil

[27] soundness.v:SemWff_cons

[28] soundness.v:SemWt_Univ

Lemma 4.9 (ST-Var)**.** *If* $\vDash \Gamma$ *and* $x : A \in \Gamma$*, then* $\Gamma \vDash x : A$.

Proof Suppose $\rho \vDash \Gamma$. By the definition of semantic typing, we need to show that there exists some i and S such that

- $[\![A\{\rho\}]\!]^i \searrow S$
- $\rho(x) \in S$

By the definition of semantic context well-formedness, we deduce from $\vDash \Gamma$ and $x : A \in \Gamma$ that there exists some universe level i such that $\Gamma \vDash A : \mathbf{Set}_i$. By the equivalence from Lemma 4.8, there exists S such that $[\![A\{\rho\}]\!]^i \searrow S$. However, by the definition of $\rho \vDash \Gamma$, we know that $\rho(x) \in S$, which is exactly what we need for the conclusion. ■

Lemma 4.10 (ST-Set)**.** *If* $i < j$*, then* $\Gamma \vDash \mathbf{Set}_i : \mathbf{Set}_j$.

Proof Immediate by Lemma 4.8 and rule IR-Set. ■

Lemma 4.11 (ST-Pi)**.** *If* $\Gamma \vDash A : \mathbf{Set}_i$ *and* $\Gamma, x : A \vDash B : \mathbf{Set}_i$*, then* $\Gamma \vDash \Pi x{:}A.B : \mathbf{Set}_i$.

Proof Applying Lemma 4.8 to the conclusion, it now suffices to show that given $\rho \vDash \Gamma$, there exists some S such that $[\![(\Pi x{:}A.B)\{\rho\}]\!]^i \searrow S$. From Lemma 4.8 and $\Gamma \vDash A : \mathbf{Set}_i$, we know that there exists some set S_0 such that $[\![A\{\rho\}]\!]^i \searrow S_0$. From $\Gamma, x : A \vDash B : \mathbf{Set}_i$, we know that there must exist S such that $[\![B\{\rho[x \mapsto a]\}]\!]^i \searrow S$ for every $a \in S_0$. The conclusion immediately follows from the admissible rule I-PiAlt. ■

Lemma 4.12 (ST-Abs)**.** *If* $\Gamma \vDash \Pi x{:}A.B : \mathbf{Set}_i$ *and* $\Gamma, x : A \vDash b : B$*, then* $\Gamma \vDash \lambda x.b : \Pi x{:}A.B$.

Proof By unfolding the definition of $\Gamma \vDash \lambda x.b : \Pi x{:}A.B$, we need to show that given some $\rho \vDash \Gamma$, there exists some i and S such that $[\![(\Pi x{:}A.B)\{\rho\}]\!]^i \searrow S$ and $(\lambda x.b)\{\rho\} \in S$.

By Lemma 4.8 and the premise $\Gamma \vDash \Pi x{:}A.B : \mathbf{Set}_i$, there exists some set S such that $[\![(\Pi x{:}A.B)\{\rho\}]\!]^i \searrow S$. It now suffices to show that $(\lambda x.b)\{\rho\} \in S$. By Lemma 3.8, the alternative inversion principle for rule I-Pi, there exists some S_0 such that all following conditions hold:

- $[\![A\{\rho\}]\!]^i \searrow S_0$
- $\forall a$, if $a \in S_0$, then $\exists S_1$, $[\![B\{\rho[x \mapsto a]\}]\!]^i \searrow S_1$
- $S = \{b \mid \forall a$, if $a \in S_0$, then $\forall S_1$, if $[\![B\{\rho[x \mapsto a]\}]\!]^i \searrow S_1$, then $b\,a \in S_1\}$

To show that $(\lambda x.b)\{\rho\} \in S$, we need to prove that given $a \in S_0$, $[\![B\{\rho[x \mapsto a]\}]\!]^i_I \searrow S_1$, we have $(\lambda x.b)\{\rho\}\, a \in S_1$. By Lemma 3.12, the set S_1 is closed under expansion. Since $(\lambda x.b)\{\rho\}\, a \Rightarrow b\{\rho[x \mapsto a]\}$, it suffices to show that $b\{\rho[x \mapsto a]\} \in S_1$, which is immediate from $\Gamma, x : A \vDash b : B$ and the fact that the logical relation is deterministic and cumulative (Lemma 3.10). ■

Lemma 4.13 (ST-App)**.** *If* $\Gamma \vDash b : \Pi x{:}A.B$ *and* $\Gamma \vDash a : A$*, then* $\Gamma \vDash b\,a : B\{a/x\}$.

Proof Suppose $\rho \vDash \Gamma$. The goal is to show that there exists some i and S_1 such that $b\{\rho\}\, a\{\rho\} \in S_1$ and $[\![B\{a/x\}\{\rho\}]\!]^i \searrow S_1$, or equivalently, $[\![B\{\rho[x \mapsto a\{\rho\}]\}]\!]^i \searrow S_1$

since $B\{a/x\}\{\rho\} = B\{\rho[x \mapsto a\{\rho\}]\}$. By the premise $\Gamma \vDash b : \Pi x{:}A.B$, Lemma 4.8, and Lemma 3.8, there exists some i and S_0 such that:

- $[\![A\{\rho\}]\!]^i \searrow S_0$
- $\forall a_0$, if $a_0 \in S_0$, then $\exists S_1$, $[\![B\{\rho[x \mapsto a_0]\}]\!]^i \searrow S_1$
- $\forall a_0$, if $a_0 \in S_0$, then $\forall S_1$, if $[\![B\{\rho[x \mapsto a_0]\}]\!]^i \searrow S_1$, then $b\{\rho\}\ a_0 \in S_1$

Instantiating the variable a_0 from the last two bullets with the term $a\{\rho\}$, the conclusion immediately follows. ■

Theorem 4.14 (The Fundamental Theorem[29])**.**

- *If* $\Gamma \vdash a : A$, *then* $\Gamma \vDash a : A$.
- *If* $\vdash \Gamma$, *then* $\vDash \Gamma$.

Proof Proof by mutual induction over the derivation of $\Gamma \vdash a : A$ and $\vdash \Gamma$. The cases related to context well-formedness immediately follow from Lemmas 4.6 and 4.7. The semantic typing rules (Lemmas 4.9, 4.10, 4.11, 4.12, 4.13) can be used to discharge their syntactic counterparts (e.g. Lemma 4.12 for case rule T-Abs). The remaining cases not covered by the lemmas are similar to the ones already shown. ■

Recall the logical consistency property (Theorem 3.1), which states that the judgment $\cdot \vdash a : \textbf{Void}$ is not derivable. We now give a proof of the property using the fundamental lemma.

Proof Suppose $\cdot \vdash a : \textbf{Void}$ is derivable, then by the fundamental lemma, we have $\cdot \vDash a : \textbf{Void}$, which states that for all $\rho \vDash \cdot$, and for all j, S such that $[\![\textbf{Void}]\!]^j \searrow S$, we have $a\{\rho\} \in S$. By Lemma 4.2, any ρ we pick trivially satisfies $\rho \vDash \Gamma$. For convenience, we pick ρ as **id**, though any ρ would work since $\cdot \vdash a : \textbf{Void}$ ensures there is no free variable in a. We have $a\{\textbf{id}\} = a \in S$. By the **Void** case of the inversion property (Lemma 3.4), we know that S must be the empty set, contradicting the assumption that $a \in S$. ■

Our soundness theorem also tells us something about closed terms of type **Bool**; they either reduce to **true** or **false**.

Corollary 4.15 (Canonicity[30])**.** *If* $\cdot \vdash b : \textbf{Bool}$, *then either* $b \Rightarrow^* \textbf{true}$ *or* $b \Rightarrow^* \textbf{false}$.

Proof The proof is similar to above, except that we use the **Bool** case of the inversion property. ■

5 Existence of β-normal forms

In this section, we show how the logical relation from Section 3 can be extended to show the existence of β normal forms for (open and closed) well-typed terms. In other words, we prove that it is possible to repeatedly use the parallel reduction relation to reduce any term

[29] `soundness.v:soundness` [30] `soundness.v:canonicity`

to its unique normal form, where no further (non-identity) reductions can be applied. This result can be used to show that our type conversion relation is decidable.

The goal of this section is also to demonstrate that our logical relations proof technique can be extended to reason about the reduction properties of open terms, not just the reduction of terms after closing substitutions. Reasoning about open terms is particularly important for dependently-typed languages because type checking involves working with open terms. While this extension employs well-known techniques, it continues to be short and demonstrates the robustness of our initial framework.

We begin this part with a description of the β-normal forms of λ^{Π}.

$$\begin{array}{llcl}
\textit{β-neutral terms} & e & ::= & x \mid e\,f \mid \mathbf{J}\,e\,f\,f\,f \mid \mathbf{if}\ e\ \mathbf{then}\ f\ \mathbf{else}\ f \\[1em]
\textit{β-normal terms} & f & ::= & e \mid \mathbf{Set}_i \mid \mathbf{Void} \mid \Pi x{:}f.f \mid f \sim f \in f \\
 & & \mid & \lambda x.f \mid \mathbf{refl} \mid \mathbf{Bool} \mid \mathbf{true} \mid \mathbf{false}
\end{array}$$

Fig. 5. β-neutral and normal forms

The syntactic forms e and f (Figure 5) capture the neutral terms and normal forms with respect to β-reduction. Instead of the metavariables e and f, we also use the judgment forms $\mathbf{ne}\ a$ and $\mathbf{nf}\ a$ to indicate that there exists e or f such that $a = e$ or $a = f$.

The predicates $\mathbf{wne}\,a$ and $\mathbf{wn}\,a$ describe terms that can evaluate into β-neutral or β-normal form through parallel reduction and are defined as follows.

$$\begin{array}{llcl}
\textit{weakly normalizes to a neutral form} & \mathbf{wne}\ a & \iff & \exists e, a \Rightarrow^{*} e \\
\textit{weakly normalizes to a normal form} & \mathbf{wn}\ a & \iff & \exists f, a \Rightarrow^{*} f
\end{array}$$

$\boxed{[\![A]\!]^{i}_{I} \searrow S}$ *(Logical Relation)*

I-Ne

$$\frac{\mathbf{ne}\ A}{[\![A]\!]^{i}_{I} \searrow \{a \mid \mathbf{wne}\ a\}}$$

I-VoidNew

$$\frac{}{[\![\mathbf{Void}]\!]^{i}_{I} \searrow \{a \mid \mathbf{wne}\ a\}}$$

I-BoolNew

$$\frac{}{[\![\mathbf{Bool}]\!]^{i}_{I} \searrow \{a \mid a \Rightarrow^{*} \mathbf{true} \lor a \Rightarrow^{*} \mathbf{false} \lor \mathbf{wne}\ a\}}$$

I-EqNew

$$\frac{\mathbf{nf}\ a \qquad \mathbf{nf}\ b \qquad \mathbf{nf}\ A}{[\![a \sim b \in A]\!]^{i}_{I} \searrow \{p \mid (p \Rightarrow^{*} \mathbf{refl} \land a \Leftrightarrow b) \lor \mathbf{wne}\ p\}}$$

Fig. 6. Extended logical relation (new and changed rules)

The updated logical relation is shown in Figure 6. [31] There is one new rule in this figure, rule I-Ne. In a non-empty context, a type itself may evaluate to a neutral term and in turn can only be inhabited by neutral terms. Otherwise, the rest of the rules in this figure are updates to the analogous rules in Figure 4. Note that, we omit the rules for the function and universe cases because they are identical to the original version.

The changes to rule I-Bool and rule I-Void follow the same pattern: an open term of type **Bool** does not necessarily reduce to **true** or **false**, but may reduce to a variable, or more generally, a neutral term. Likewise, while the **Void** type remains uninhabited under an empty context, it may be inhabited when there is a variable in the context that has type **Void** or that can be eliminated to type **Void**.

The rule for equality type $a \sim b \in A$ is augmented with the precondition that a, b, and A are all normal forms because otherwise our model would include equality types that are themselves not normalizing. Furthermore, the condition $a \Leftrightarrow b$ is only required when the equality proof reduces to **refl**. If the proof term reduces to a neutral term, then there is nothing we need to show about the relationship between a and b.

Because we are working with open terms, we need a few additional syntactic lemmas about reduction. First, a renaming ξ is a generalization of weakening when working with simultaneous substitutions. It consistently maps the variables that appears in terms to other variables. If a renamed term has been reduced, we can always recover the result of the reduction without the renaming.

Lemma 5.1 (Par anti-renaming[32])**.** *If $a\{\xi\} \Rightarrow b_0$, then there exists some b such that $b\{\xi\} = b_0$ and $a \Rightarrow b$.*

We can show that parallel reduction preserves β-normal and neutral forms.

Lemma 5.2 (Par preserves β-neutral and normal forms[33])**.** *If $a \Rightarrow b$, then*

- **ne** *a implies* **ne** *b*
- **nf** *a implies* **nf** *b*

Lemma 5.2 could have been strengthened to say that if **ne** a or **nf** a and $a \Rightarrow b$, then $a = b$. Since **ne** and **nf** captures terms free of β redexes, parallel reduction cannot take any real reduction steps and can only step into a term itself. However, for the purpose of our proof, Lemma 5.2 is sufficient.

All the properties we have shown in Section 3 and 4 before the fundamental lemma can be proven in the same order, where the new cases due to rule I-Ne and the augmentation of neutral terms to rules I-Void, I-Eq, and I-Bool can be immediately discharged by Lemma 5.2.

Furthermore, Lemma 5.2, in its current weaker form, would still hold after we extend our equational theory with the function η rule, where parallel reduction can take η steps but still preserves β-normal form.

[31] `semtypingopen.v:InterpExt`

[32] `normalform.v:Par_antirenaming`

[33] `normalform.v:nf_ne_preservation`

We also need to know that the **wne** a and **wn** a relations can be justified compositionally. For example, an application has a neutral form when the function has a neutral form and the argument has a normal form.

Lemma 5.3 (Wne application[34])**.** *If* **wne** a *and* **wn** b*, then* **wne** $(a\,b)$.

Proof Immediate by induction over the length of the reduction sequences in **wne** a and **wn** b. ■

Furthermore, if we know that an application of a term to a variable has a normal form, then we know that the term must have a normal form.

Lemma 5.4 (Wn extensionality[35])**.** *If* **wn** $(a\,x)$*, then* **wn** a.

Proof By induction over the length of the reduction sequence in **wn** $(a\,x)$. The conclusion follows from Lemmas 5.1 and 5.2. ■

Before we can prove the fundamental theorem and derive the normalization property as its corollary, we need to additionally formulate and prove an *adequacy property* about the logical relation. This property, that the interpretation of each type is a *reducibility candidate*, allows us to conclude that every term in each interpretation has a normal form. In the previous section, we only needed a property of the interpretation of the **Void** type. However, for this section, we need to know something about the interpretation of every type.

Furthermore, to prove this adequacy property, we need to strengthen it to also give us more information about neutral terms as we proceed by induction. In particular, we need to know that all terms that reduce to neutral forms are contained within the interpretation. Therefore, we formally define when a set is a *reducibility candidate* (shortened as CR) as follows. Our definition of CR is inspired by Girard et al. (1989), but not identical since we only care about weak normalization.

Definition 5.5 (Reducibility Candidates (CR)[36])**.** Let S be a set of terms. We say that $S \in CR$ if and only if conditions CR_1 and CR_2 hold.

- $S \in CR_1 \iff \forall a$, if **wne** a, then $a \in S$
- $S \in CR_2 \iff \forall a$, if $a \in S$, then **wn** a

We now state and prove the adequacy lemma.

Lemma 5.6 (Adequacy[37])**.** *If* $\llbracket A \rrbracket^i \searrow S$*, then we have* $S \in CR$.

Proof We start by strong induction over i. We are given the induction hypothesis that for all $j < i$, $\llbracket A \rrbracket^j \searrow S$ implies $S \in CR$. Our goal is to show $\llbracket A \rrbracket^i \searrow S$ implies $S \in CR$.

By Definition 3.2, we have the equality $\llbracket A \rrbracket^i \searrow S = \llbracket A \rrbracket^i_I \searrow S$ where $I(i) := \{A \mid \exists S, \llbracket A \rrbracket^i \searrow S\}$. We then proceed by structural induction over the derivation of $\llbracket A \rrbracket^i_I \searrow S$. The

[34] `normalform.v:wne_app`

[35] `normalform.v:ext_wn`

[36] `semtypingopen.v:CR`

[37] `semtypingopen.v:adequacy`

only interesting cases are rule I-Pi and rule I-Set. The function case requires Lemmas 5.4 and 5.3, which we have shown earlier.

The rule I-Set case is the most interesting. We must show that for all $j < i$, the set $\{A \mid \exists S, \llbracket A \rrbracket^j \searrow S\} \in CR$. We immediately know that $\{A \mid \exists S, \llbracket A \rrbracket^j \searrow S\} \in CR_1$ by rule I-Ne. It remains to show that $\{A \mid \exists S, \llbracket A \rrbracket^j \searrow S\} \in CR_2$, or equivalently, for all A, $\llbracket A \rrbracket^j \searrow S$ implies $\mathbf{wn}\, A$. Suppose $\llbracket A \rrbracket^j \searrow S$ for an arbitrary A. We have $\llbracket A \rrbracket^j \searrow S = \llbracket A \rrbracket^j_I \searrow S$ where I has the same definition from earlier but its domain restricted to numbers less than j. We perform another induction on the derivation of $\llbracket A \rrbracket^j_I \searrow S$. All cases are trivial except for the case for rule I-Pi. Our induction hypothesis immediately gives us $\mathbf{wn}\, A$. To derive $\mathbf{wn}\, (\Pi x{:}A.B)$, it remains to show $\mathbf{wn}\, B$. We use the outermost induction hypothesis to show that x semantically inhabits A, from which we derive $\mathbf{wn}\, (B\{x/x\})$ and conclude $\mathbf{wn}\, B$ through antirenaming (Lemma 5.1). ■

The formulation of semantic well-typedness and the fundamental lemma from Section 4 remains unchanged. The proof of the fundamental lemma[38] is still carried out by induction over the typing derivation, where the additional neutral term related cases are handled by Lemma 5.6, the adequacy property.

The normalization property then follows as a corollary of the fundamental theorem.

Corollary 5.7 (Existence of β-normal forms[39]). *If* $\Gamma \vdash a : A$, *then* $\mathbf{wn}\, a$ *and* $\mathbf{wn}\, A$.

Proof By the fundamental lemma, we know that $\Gamma \vDash a : A$. That is, for all $\rho \vDash \Gamma$, there exists some i and S such that $\llbracket A\{\rho\} \rrbracket^i \searrow S$ and $a\{\rho\} \in S$. We pick the ρ to be the identity substitution $\mathbf{id}$, which injects variables as terms. The side condition $\mathbf{id} \vDash \Gamma$ is satisfied since Lemma 5.6 says neutral terms, including variables, semantically inhabit any S_0 where S_0 is the interpretation of some type. With our choice of ρ, we have $A\{\rho\} = A\{\mathbf{id}\} = A$ and $a\{\rho\} = a\{\mathbf{id}\} = a$. Then we know that $\llbracket A \rrbracket^i \searrow S$ and $a \in S$ for some i and S. By Lemma 5.6, we conclude that $\mathbf{wn}\, a$ and $\mathbf{wn}\, A$. ■

The extension of our logical relation to prove normalization of open *and* closed terms closely mirrors the progression from normalization of closed terms (Harper, 2022*a*) to normalization of open terms (Harper, 2022*b*) in the simply typed lambda calculus. Indeed, a mechanization of normalization generalized to open terms appears in Abel et al. (2019). In this setting, as above, adequacy must be proven before the fundamental theorem so they can handle elimination rules such as rule T-App where the scrutinee is a neutral term. Dependent types make the adequacy proof slightly more complicated because we also need to know that every *type* has a normal form, not just terms. This complicates our proof specifically in the rule I-Set case for our adequacy property (Lemma 5.6).

Overall, despite the dependently typed setting, it is in fact reassuring that once we have laid the foundational technique for handling dependent types in our logical relation, the extension to open terms mostly boils down to properties that can be independently derived from the logical relation through syntactic means.

[38] `soundnessopen.v:soundness`

[39] `soundnessopen.v:mltt_normalizing`

6 Existence of $\beta\eta$-normal forms

Wieczorek and Biernacki (2018); Abel et al. (2017); Adjedj et al. (2024) include the η law for functions in their equational theory and use relational models to justify its validity. In our system, we can easily incorporate the function η law to the equational theory of λ^{Π} by adding the following parallel reduction rule.

$$\textsc{P-AbsEta}\quad \frac{y \notin \mathbf{fv}(a_0) \qquad a \Rightarrow a_0}{\lambda y.((\lambda x.a)\ y) \Rightarrow a_0}$$

In this section, we show how we easily extend the existence of β-normal forms from Section 5 to the existence of $\beta\eta$-normal forms after this addition.

First, we recover the same confluence result about parallel reduction using the standard techniques from Barendregt (1993); Takahashi (1995), though anti-renaming (Lemma 5.1) must be proven before the diamond property (Lemma 2.5). Another complication is that the anti-renaming property and the diamond property for parallel reduction are now proven through induction on a size metric of lambda terms; rule P-AbsEta reduces a term that is not a strict subterm.

Note that, after this extension, the specification of our logical relation does not require any updates. The proof of the fundamental theorem also remains identical since the complications introduced by η are hidden behind the proofs of the diamond property and the anti-renaming property. As before, **ne** and **nf** represent β-neutral and β-normal forms, and the fundamental lemma shows us that every well-typed term has a β-normal form. However, in the presence of the η reduction rule, Lemma 5.2 tells us that η reduction preserves β-normal forms (i.e. does not produce new β-redexes). Furthermore, since the η reduction rule for functions strictly decreases the size of the term, the existence of $\beta\eta$ normal form trivially follows.

Corollary 6.1 (Existence of $\beta\eta$-normal form)**.** *If $\Gamma \vdash a : A$, then a has $\beta\eta$-normal form.*

A well-known issue with our approach is the failure of syntactic confluence when the lambda term contains type annotations. A simple counterexample is $\lambda y{:}B.((\lambda x{:}A.a)\ y)$ where $y \notin \mathbf{fv}((\lambda x{:}A.a))$; depending on whether rule P-AbsEta is performed on the whole term or rule P-AppAbs is used on the inner β redex, we end up with the terms $\lambda x{:}B.a$ (after α-conversion) or $\lambda x{:}A.a$, where A and B are not necessarily syntactically equal terms. Choudhury et al. (2022) resolve this problem by stating their confluence result in terms of an equivalence relation that quotients out parts of the terms that are computationally irrelevant; the annotations of lambda terms are ignored since the behavior of a lambda term is not affected by its type annotation. We believe the same approach is applicable to our proof.

The bigger issue is extensions such as η-laws for unit and products. Surjective pairing, for example, is not confluent for untyped lambda terms Klop and de Vrijer (1989). The relational, type-annotated, and Kripke-style models from Wieczorek and Biernacki (2018); Abel et al. (2017); Adjedj et al. (2024) can be more easily extended to support these rules. We note, however, that the issue with η rules is not exclusive to dependently typed languages and has been studied in more limited languages that are either simply typed (Pierce, 2004;

Pfenning, 1997) or dependently typed but without large eliminations (Harper and Pfenning, 2005; Abel and Coquand, 2005). Common workarounds include type-directed conversion and shifting the focus to obtaining η-long forms Abel and Scherer (2012).

While not without limitations, our simple proof demonstrates the core building blocks of more complex arguments, thus paving the way for experimentation and eventual extension to more expressive systems.

7 Mechanization

	Consistency	Normalization	Syntactic metatheory
Syntactic typing	83	=	=
Untyped reduction	344	=	=
Neutral and normal forms	-	273	-
Logical relation	338	430	-
Semantic soundness	192	211	-
Syntactic soundness	-	-	629
Total	957	1341	1056

Fig. 7. Nonblank, noncomment lines of code of the Coq Development. The marker = indicates that the line count is the same as the column to the left. The marker - indicates the file does not contribute to the total.

To demonstrate the scale of our proof scripts, Figure 7 shows the number of non-blank, non-comment lines of code[40] for each file of our development, including the base consistency proof from Section 3 and 4 and the extension to β-normalization from Section 5. For comparison, we have also proven syntactic type soundness through preservation [41] and progress[42].

The $\beta\eta$-normalization proof from Section 6 comprises 1568 lines of non-blank, non-comment lines of code. We choose not to include it in the chart, because of slight differences in lemma dependencies for untyped reduction and normal forms that make the comparison less informative. However, when compared to the β-normalization extension, the $\beta\eta$ extension has the same line count in the definition of the logical relation and the semantic soundness proof.

The Autosubst 2 tool takes our 13 line syntax specification, written in higher-order abstract syntax, and generates the Coq syntax specification, renaming and substitution functions, and lemmas and tactics that allow reasoning about those functions. The auto-generated syntax file (291 LOC) and other Autosubst library files are also not included in the figure.

7.0.0.1 Axioms. Our Coq development assumes two axioms: functional extensionality and propositional extensionality. The former is also required by the Autosubst 2 libraries. Both axioms are known to be consistent with Coq's metatheory. These axioms bridge the gap

[40] calculated by the `tokei` tool, available from `https://github.com/XAMPPRocky/tokei`.

[41] `syntactic_soundness.v:subject_reduction`

[42] `syntactic_soundness.v:wt_progress`

between our mechanization and our informal proofs. For example, in set theory, to show that two sets S_0 and S_1 are equal, it suffices to show the extensional property that $\forall x, x \in S_0 \iff x \in S_1$. We leverage this fact occasionally in our presented proofs. However, in Coq, sets of terms ($\mathcal{P}(Term)$) are encoded as the type `tm -> Prop`, a predicate over λ^Π terms. In axiom-free Coq, predicates do not come with the extensionality property. Given two predicates P and Q, we cannot conclude that $P = Q$ when given a proof of $\forall x, P(x) \iff Q(x)$. But this is exactly the statement of predicate extensionality, an immediate corollary from functional extensionality and propositional extensionality.

7.0.0.2 Encoding the logical relation in Coq.

We next discuss specific details of the Coq encoding of the logical relation presented in Section 3.

In the Coq mechanized proof, the definition of $[\![A]\!]^i_I \searrow S$ has type `Prop`, where `I` has type `nat -> tm -> Prop` and `S` has type `tm -> Prop`.

However, if desired, we could consistently replace the use of `Prop` with Coq's predicative sort `Type` in the definition of $[\![A]\!]^i_I \searrow S$. This alternative definition could be part of the interpretation for any *finite* number of universes. The use of `Type` becomes troublesome only when we attempt to define $[\![A]\!]^i \searrow S$, the top-level logical relation (Definition 3.2) that recursively calls itself at smaller universe levels. Therefore, the one feature of λ^Π that truly requires impredicativity is its countable universe hierarchy.

The definition of $[\![A]\!]^i_I \searrow S$ has an almost one-to-one correspondence to the Coq definition. The main difference is the specification of I. In Section 3, we define I as a function over numbers less than i, the universe level. In Coq, we only require I to be a function with the set of natural numbers as its domain. In the Coq encoding of $[\![A]\!]^i \searrow S$, we define $I \in \mathbb{N} \to \mathcal{P}(Term)$ as follows.

$$I(j) = \begin{cases} \{A \mid \exists S, [\![A]\!]^j \searrow S\} & \text{when } j < i \\ \emptyset & \text{otherwise} \end{cases}$$

Since I is only applied to numbers strictly less than i in rule I-Set, we can retroactively show that the set we return in the $j \geq i$ case is junk data that does not affect the result of the logical relation. This property allows us to recover the simple equation for $[\![A]\!]^i \searrow S$ shown in Definition 3.2.

Rule I-PiCoq shows how rule I-Pi is actually encoded in our mechanized proof.

I-PiCoq

$$\frac{\begin{array}{c} [\![A]\!]^i_I \searrow S \qquad R \in S \times \mathcal{P}(Term) \\ \forall a, \exists S_0, (a, S_0) \in R \qquad \forall a, \forall S_0, \text{if } (a, S_0) \in R, \text{then } [\![B\{a/x\}]\!]^i_I \searrow S_0 \end{array}}{[\![\Pi x{:}A.B]\!]^i_I \searrow \{b \mid \forall a, \forall S_0, \text{if } (a, S_0) \in R, \text{then } b\ a \in S_0\}}$$

Compared to rule I-Pi, rule I-PiCoq replaces the function F with a total relation R. The equivalence of these two rules follows from the fact that the logical relation is a partial function (Lemma 3.7). In set-theoretic notation, rule I-Pi is more readable. However, if we want to encode the same rule in Coq, we must encode F as a relation (with type `tm -> (tm -> Prop) -> Prop`) that satisfies the functionality constraint: `forall a S0 S1, F a S0 -> F a S1 -> S0 = S1`. In comparison, rule I-PiCoq does not require this side condition and results in a simpler definition.

We note that we cannot ascribe F the type `tm -> (tm -> Prop)` since Coq requires functions of such type to be computable. While defining F as a computable Coq function rather than a functional relation does result in a concise encoding of rule I-Pi, we will have trouble instantiating F with the logical relation, which is defined as a relation that we prove to be functional, rather than a computable function.

7.0.0.3 Automation. Our Coq mechanization heavily uses automation, supported by the tools Autosubst 2 (Stark et al., 2019) and CoqHammer (Czajka and Kaliszyk, 2018).

We use the Autosubst 2 framework to produce Coq syntax files based on a de Bruijn representation of variable binding and capture-avoiding substitution. In addition to these generated definitions, Autosubst 2 provides a powerful tactic `asimpl` that can be used to prove the equivalence of two terms constructed using the primitive operators provided by the framework. This tactic simplifies the reasoning about substitution as many substitution-related properties about syntax are immediately discharged by `asimpl`.

For other automation tasks that are not specific to binding, we use the powerful `sauto` tactic provided by CoqHammer to write short and declarative proofs. For example, here is a one-line proof of the triangle property about parallel reduction, from which the diamond property (Lemma 2.5) follows as a corollary. The triangle property states that if $a \Rightarrow b$, then $b \Rightarrow a^*$, where a^* is the Takahashi translation (Takahashi, 1995) which roughly corresponds to simultaneous reduction of the redexes in a, excluding the new redexes that appear as a result of reduction.

```
Lemma Par_triangle a : forall b, (a ⇒ b) -> (b ⇒ tstar a).
Proof.
  apply tstar_ind; hauto lq:on inv:Par use:Par_refl,Par_cong ctrs:Par.
Qed.
```

In prose, the triangle property can be proven by induction over the graph of `tstar a`, the Takahashi translation. Options `inv:Par` and `ctrs:Par` say that the proof involves inverting and constructing of the derivations of parallel reduction. The option `use:Par_refl,Par_cong` allows the automation tactic to use the reflexivity and congruence properties of parallel reduction as lemmas.

The flag `lq:on` tunes CoqHammer's search algorithm. While this flag appears arcane, when developing our proof scripts we never specify this option manually. Instead, we first invoke the `best` tactic provided by CoqHammer, specifying only the `inv`, `ctrs`, and lemmas that we want to use. The `best` tactic then iterates through possible configurations and provides us with a replacement with the tuned performance flags that save time for future re-execution of the proof script.

The automation provided by CoqHammer not only gives us a proof that is shorter and more resilient to changes, but also provides useful documentation for readers who wish to understand the mechanized proof. Although automation performs extensive search, we can configure it to not use lemmas or invert derivations that are not specified in the `use` or `inv` flags.

8 Related Work

8.1 Logical relations for dependent types

In the most general sense, a logical relation can be viewed as a practical technique that uses a type-indexed relation to strengthen the induction hypothesis for the property of interest. The original idea of this technique can be traced back to Tait (1967). This proof maps types to sets of terms satisfying certain properties related to reduction. The same idea is explained in Girard et al. (1989) and extended to prove strong normalization of System F. Tait's method has also been successfully applied to dependently typed languages to prove strong normalization Martin-Löf (1975); Luo (1990); Geuvers (1994); Barendregt (1993).

However, the pen-and-paper representation of logical relations proofs can be challenging to adapt to a theorem prover since many details are hidden behind concise notations. For example, Geuvers (1994) presents the interpretation for types as an inductively defined total function over the set of syntactically well-formed types. In untyped set theory, it makes sense to define the logical relation as a simply-typed function that takes a type and returns some set; however in constructive type theory, the metalogic of Coq and Agda, the interpretation function must be a dependently-typed function whose return type depends on the derivation of the well-typedness of its input. The well-typedness derivation and the proof of the classification theorem are examined in the body of the interpretation function to decide whether an argument of an application should be erased during interpretation. As a result, this definition causes difficulties for modern proof assistants. Due to the impredicativity of the object language, Geuvers (1994)'s proof cannot be encoded in Agda, which has a predicative metatheory. Due to the use of proof-relevant derivations, even in Coq, a proof assistant that supports impredicativity, one would need to constantly juggle between the impredicative but irrelevant sort `Prop` sort and the predicative but relevant sort `Type`.

More recent work such as Abel and Scherer (2012) and Abel et al. (2008) make their definitions more explicit and precise and thus more directly encodable in proof assistants. Our logical relation resembles their definition of a semantic universe hierarchy, although we close our relation under expansion with respect to parallel reduction rather than weak-head reduction. Furthermore, Abel and Scherer (2012) and Abel et al. (2008) use their semantic universe hierarchy as a measure to define Kripke-style logical relations, from which they derive the correctness of their conversion algorithms. In our work, we use the semantic universe hierarchy directly in our definition of semantic typing because it is sufficient for our purposes (consistency and normalization).

8.2 Mechanized logical relations for dependent types

	U	Ind	C	L E	A	Main results
λ^{Π} (this work)	$\mathbb{N}$	Id, Bool	U	✓	1	Consistency and normalization
λ^{θ}	0	Id, Nat	U	%	1	Consistency
Core Nuprl	$\mathbb{N}$	W-Types	E	✓	2	Consistency
NBE-in-Coq	1	Nat	T	%	2	Correctness of NBE
$\lambda^{\Pi U \mathbb{N}}$	1	Nat, Σ	T	✓	2	Decidability of conversion
MLTT-á-la-Coq	1	Id, Nat, Σ	T	✓	2	Decidibility of type checking

Universes:	Countable ($\mathbb{N}$), Zero (0), One (1)
Inductives:	Identity types (Id), Natural numbers (Nat), Σ-types (Σ), W-types
Conversion:	Untyped (U), Typed (T), Extensional (E)
Large Eliminations:	Included (✓), not included (%)
Arity of interpretation:	Sets of terms (1), Relations between terms (2)

λ^{θ}	Casinghino et al. (2014) (logical fragment only)
Core Nuprl	Anand and Rahli (2014)
$\lambda^{\Pi U \mathbb{N}}$	Abel et al. (2017)
NBE-in-Coq	Wieczorek and Biernacki (2018)
MLTT-á-la-Coq	Adjedj et al. (2024)

Fig. 8. Feature matrix for dependently typed languages with mechanized logical relations

Figure 8 presents several mechanized proofs that feature logical-relations arguments for dependently-typed languages. Each of these proofs is significantly larger than than our development; but they also prove more results about different object languages.The table provides a comparison between the various features of their object languages, but is not exhaustive. For example, Casinghino et al. (2014) and Anand and Rahli (2014) both have support for partial programs. However, we include features that we believe to be most impactful to the definition of the logical relation.

Casinghino et al. (2014) introduce λ^{θ}, a dependently typed programming language that uses modality to distinguish between logical proofs and programs. The consistency proof of λ^{θ}'s logical fragment has been mechanized in Coq through a step-indexed logical relation; step-indexing is required to model the programmatic fragment, which interacts with the logical fragment. The lack of polymorphism and type-level computation means their logical relation can be defined recursively for well-formed types using a size metric, which has been used in Liu and Weirich (2023). Their development is around 8,000 lines of nonblank, noncomment code.

Abel et al. (2017) mechanize in Agda the decidability of type conversion rule for a dependently typed language with one predicative universe level and a typed judgmental equality that includes the function η law. They use a Kripke-style logical relation parameterized over a type-directed equivalence relation satisfying certain properties to facilitate the reuse of their definition. The logical relation is defined using the induction-recursion scheme, which is available in Agda but not in Coq. Their development involves around 10,000 lines of Agda code. Adjedj et al. (2024) transports the logical relation from Abel et al. (2017) in the predicative fragment of Coq and further extends the decidability of type conversion result from Abel et al. (2017) to the decidability type checking of a bidirectional type system. Their development has around 30,000 lines of Coq code.

Anand and Rahli (2014) mechanize the metatheory of Nuprl (Constable et al., 1986) in Coq. This metatheory is an extensional type theory with features such as dependent functions, inductive types, partial types, and a full universe hierarchy. They construct a PER model in Coq to show the logical consistency of their language. Their development has been further extended with features such as intersection types, union types, and quotient types. The extensive coverage of features results in a Coq development with around 330,000 lines of code. Wieczorek and Biernacki (2018) mechanize the normalization-by-evaluation algorithm in Coq for a dependently typed language with one predicative universe, similar to Abel et al. (2017) and Adjedj et al. (2024). However, since their type system has no elimination form for natural numbers, the only base type from the object language, large elimination is not supported despite the one predicative universe. Their development has around 20,000 lines of Coq code. Both Anand and Rahli (2014) and Wieczorek and Biernacki (2018) leverage the impredicative `Prop` sort of Coq to define the interpretation of dependent function types and thus are closely related to our mechanization. Anand and Rahli (2014) further show it is possible to encode a finite universe hierarchy without the use of either impredicativity or induction-recursion. Their encoding of a countable universe hierarchy relies on impredicativity, similar to our development.

8.3 Other mechanized metatheory of dependent types

Barras (2010); Wang and Barras (2013) assign set-theoretic semantics to dependent type theory in Coq. Unlike the previous efforts, which primarily focus on predicative type theory and more direct reducibility models, Barras (2010); Wang and Barras (2013) tackle extensions of CC^{ω}, a system that incorporates a predicative universe on top of the impredicative sort in the Calculus of Constructions. We choose to focus on a syntactic term model so we do not have to take the extra step of mechanizing mathematical objects such as sets and domains.

There are other mechanized developments for dependently typed systems that only involve properties that are derivable through syntactic means. For example, Sozeau et al. (2019) prove the correctness of a type checker for the Predicative, Cumulative Calculus of Inductive Constructions (PCUIC), Coq's core calculus, assuming the strong normalization property of the object language. Weirich et al. (2017) define System D, a core calculus of dependent Haskell, and prove the syntactic type soundness of the type system. Because System D includes nontermination, they proved the consistency of definitional equality from the confluence of parallel reduction.

Compared to the systems described here, the most notable features we are missing are cumulativity and impredicativity. Our semantic model already satisfies the cumulativity property (Lemma 3.9), but we need to extend our convertibility relation into a subtyping relation in our syntactic typing rules. Impredicativity, on the other hand, is known to be difficult to model when the impredicative sort is at the bottom of a predicative universe hierarchy; in this scenario, the erasure technique from Geuvers (1994) is not applicable (Abel, 2013). Whether there is a similarly short and simple treatment for impredicativity remains an open question.

9 Discussion

Our short consistency proof achieves the goal of demonstrating the technique of proof by logical relation for dependently typed languages. However, what remains unanswered is what makes our development significantly shorter. Are we proving simpler results for smaller languages, or making more use of automation, or is our proof technique genuinely more efficient?

First, the metatheoretic properties that we prove are indeed simpler. Compared to Core Nuprl, our system lacks extensionality, which would require a relational model to justify consistency. Because the conversion rule for λ^{Π} is untyped, we do not need a Kripke-style relational model to prove Π-injectivity among other properties, unlike systems with typed conversion. Furthermore, we prove the existence of normal forms, which induces a simple *normalize-and-compare* procedure for type conversion Pierce (2004). Wieczorek and Biernacki (2018); Abel et al. (2017), on the other hand, need to show how their algorithmic conversion procedure is sound and complete with respect to their respective declarative equational theory.

Second, the definition of our logical relation does contribute to a more concise proof. In rules I-Red and I-Bool, we choose parallel reduction, a full reduction relation, to close over our semantic interpretation of types and terms. Parallel reduction is non-deterministic, but it satisfies useful structural properties such as congruence (Lemma 2.3) and the diamond property (Lemma 2.5). We pay the price of using a non-deterministic reduction relation when we want to prove that our logical relation is a partial function; because of rule I-Red, we can have $A \Rightarrow B_0$ and $A \Rightarrow B_1$, where B_0 and B_1 each have their separate interpretations that we have to prove to be equal. Fortunately, this complexity is reconciled by the diamond property, which is easy to derive syntactically.

In contrast, Abel et al. (2017) and Wieczorek and Biernacki (2018) employ a deterministic weak head reduction relation. A deterministic reduction relation makes the functionality of a logical relation trivial to prove, but fails to satisfy the substitution property (Lemma 2.4), an issue that has been observed by Casinghino et al. (2014). If we had chosen to work with a deterministic reduction relation, we would likely need results such as the factorization theorem (Takahashi, 1995; Accattoli et al., 2019) in our development before we can prove the fundamental theorem, leading to a more complicated proof.

With untyped conversion, we sidestep the relational, Kripke-style logical relation found in other mechanized proofs. However, our early dependence on confluence before the fundamental theorem is established can be alarming. In a system with type-directed reduction, confluence is not immediately available because it depends on Π-injectivity, which is usually only proven after the fundamental theorem. Fortunately, there are syntactic workarounds for the Π-injectivity problem that allow us to recover the confluence property independently from the logical relation. Siles and Herbelin (2012) generalize the notion of

Type Parallel One Step Reduction from Adams (2006) to syntactically prove Π-injectivity for arbitrary Pure Type Systems. Weirich et al. (2017) add Π-injectivity to their equational theory, thus allowing subject reduction to be proven independently from confluence. By adopting these techniques that allow us to derive confluence early even for systems with type-directed reduction, we believe our proof technique can significantly shorten the existing logical relation proofs for systems with typed judgmental equality. We leave that as part of our future work.

10 Conclusion

In this work, we present a short and mechanized proof by logical relations for a dependently typed language with a full universe hierarchy, large eliminations, an intensional identity type, and dependent eliminators. We show the extensibility of our approach by proving the existence of $\beta\eta$-normal forms with only small and mechanical changes to our proof development. Our Coq mechanization leverages existing Coq libraries for reasoning about metatheory and for general purpose automation, allowing us to significantly reduce the verbosity typically associated with mechanized proofs. The result is a declarative proof style that rivals pen and paper.

Related work gives us confidence that we could extend our logical relation to include features such as full inductive datatypes, irrelevant arguments, and type-directed conversion; however, it is not clear how much of the brevity of this development can be maintained. Furthermore, we hope that mechanized logical relations proofs will eventually grow to include other features found in dependent type theories, such as impredicative universes, universe polymorphism, and cumulativity. Regardless, our development shows that proofs by logical relations for dependent types are accessible and do not require months of effort to implement. We hope our proof can inspire researchers to more frequently mechanize results, such as consistency and normalization, for their dependent type theories.

References

Andreas Abel. 2013. Normalization by evaluation: Dependent types and impredicativity. *Habilitation. Ludwig-Maximilians-Universität München* (2013).

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.

Andreas Abel and Thierry Coquand. 2005. Untyped Algorithmic Equality for Martin-Löf's Logical Framework with Surjective Pairs. In *Typed Lambda Calculi and Applications*, Paweł Urzyczyn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 23–38.

Andreas Abel, Thierry Coquand, and Peter Dybjer. 2008. Verifying a Semantic $\beta\eta$-Conversion Test for Martin-Löf Type Theory. In *Mathematics of Program Construction*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 29–56.

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2017. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.* 2, POPL, Article 23 (dec 2017), 29 pages. https://doi.org/10.1145/3158111

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1 (2012), 1:29. https://doi.org/10.2168/LMCS-8(1:29)2012

Beniamino Accattoli, Claudia Faggian, and Giulio Guerrieri. 2019. Factorization and Normalization, Essentially. In *Programming Languages and Systems*, Anthony Widjaja Lin (Ed.). Springer International Publishing, Cham, 159–180.

Robin Adams. 2006. Pure type systems with judgemental equality. *Journal of Functional Programming* 16, 2 (2006), 219–246.

Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. 2024. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs* (London, UK) *(CPP 2024)*. Association for Computing Machinery, New York, NY, USA, 230–245. https://doi.org/10.1145/3636501.3636951

Abhishek Anand and Vincent Rahli. 2014. Towards a formally verified proof assistant. In *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings 5*. Springer, 27–44.

Henk Barendregt. 1991. Introduction to generalized type systems. *Journal of Functional Programming* 1, 2 (1991), 462–490. https://doi.org/10.1017/S0956796800020025

Hendrik Pieter Barendregt. 1985. *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.

Henk P. Barendregt. 1993. *Lambda Calculi with Types*. Oxford University Press, Inc., USA, 117–309.

Bruno Barras. 2010. Sets in Coq, Coq in sets. *Journal of Formalized Reasoning* 3, 1 (2010), 29–48.

Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, step-indexing and compiler correctness. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/1596550.1596567

William J. Bowman and Amal Ahmed. 2015. Noninterference for free. *SIGPLAN Not.* 50, 9 (aug 2015), 101–113. https://doi.org/10.1145/2858949.2784733

Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining Proofs and Programs in a Dependently Typed Language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 33–45. https://doi.org/10.1145/2535838.2535883

Pritam Choudhury, Harley Eades III, and Stephanie Weirich. 2022. A Dependent Dependency Calculus. In *Programming Languages and Systems, ESOP 2022 (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 403–430. https://doi.org/10.1007/978-3-030-99336-8_15 Artifact available.

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. 1986. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., USA.

Coq Development Team. 2019. *The Coq Proof Assistant*. https://doi.org/10.5281/zenodo.3476303

Thierry Coquand and Christine Paulin. 1990. Inductively defined types. In *COLOG-88*, Per Martin-Löf and Grigori Mints (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–66. https://doi.org/10.1007/3-540-52335-9_47

Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for dependent type theory. *Journal of automated reasoning* 61 (2018), 423–453.

Nicolaas Govert de Bruijn. 1994. Some extensions of Automath: the AUT-4 family. In *Studies in Logic and the Foundations of Mathematics*. Vol. 133. Elsevier, 283–288.

Herman Geuvers. 1994. A short and flexible proof of strong normalization for the calculus of constructions. In *International Workshop on Types for Proofs and Programs*. Springer, 14–38.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Vol. 7. Cambridge university press Cambridge.

Robert Harper. 2016. *Practical foundations for programming languages*. Cambridge University Press.

Robert Harper. 2022a. How to (Re)Invent Tait's Method. (2022).

Robert Harper. 2022b. Kripke-Style Logical Relations for Normalization. (2022).

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

J.W. Klop and R.C. de Vrijer. 1989. Unique normal forms for lambda calculus with surjective pairing. *Information and Computation* 80, 2 (1989), 97–113. https://doi.org/10.1016/0890-5401(89)90014-X

Yiyun Liu and Stephanie Weirich. 2023. Dependently-Typed Programming with Logical Equality Reflection. *Proceedings of the ACM on Programming Languages* 7, ICFP (2023), 649–685.

Zhaohui Luo. 1990. *An extended calculus of constructions*. Ph. D. Dissertation. University of Edinburgh.

Per Martin-Löf. 1975. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73, Proceedings of the Logic Colloquium*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. North-Holland, 73–118. https://doi.org/10.1016/S0049-237X(08)71945-1

James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 128–148.

Frank Pfenning. 1997. Computation and deduction. *Unpublished lecture notes* 277 (1997).

Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

Benjamin C Pierce. 2004. *Advanced topics in types and programming languages*. MIT press.

Andrew M. Pitts. 1998. Existential types: Logical relations and operational equivalence. In *Automata, Languages and Programming*, Kim G. Larsen, Sven Skyum, and Glynn Winskel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 309–326.

Vincent Siles and Hugo Herbelin. 2012. Pure type system conversion is always typable. *Journal of Functional Programming* 22, 2 (2012), 153–180.

Lau Skorstengaard. 2019. An Introduction to Logical Relations. arXiv:1907.11133 [cs.PL]

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 8 (dec 2019), 28 pages. https://doi.org/10.1145/3371076

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* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 166–180. https://doi.org/10.1145/3293880.3294101

W. W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *The Journal of Symbolic Logic* 32, 2 (1967), 198–212. http://www.jstor.org/stable/2271658

M. Takahashi. 1995. Parallel Reductions in λ-Calculus. *Information and Computation* 118, 1 (1995), 120–127. https://doi.org/10.1006/inco.1995.1057

Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2022. *Programming Language Foundations in Agda*. https://plfa.inf.ed.ac.uk/22.08/

Qian Wang and Bruno Barras. 2013. Semantics of Intensional Type Theory extended with Decidable Equational Theories. In *Annual Conference for Computer Science Logic*. https://api.semanticscholar.org/CorpusID:16825742

Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110275

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* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 266–279. https://doi.org/10.1145/3167091