JFP, 37 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)

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 a propositional equality 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 predicative universe hierarchy, large elimination, a propositional equality 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 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 may have the impression that logical relations are not applicable to dependent types.

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 rely on logical relations 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.

Our paper instead provides a gentle and accessible introduction to a powerful technique for dependent type theories. It is accompanied by a short mechanized proof development of fewer 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 and 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 extend our consistency proof to show the existence of $\beta\eta$-normal forms for well-typed closed *and* open terms, at a moderate cost of 600 lines of code. We include a full predicative universe hierarchy with large elimination to demonstrate the logical strength of the approach.

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 inductively, prove its functionality, define semantic typing in terms of the relation, and prove the fundamental theorem, from which consistency follows as a corollary (Section 4). Our proof showcases the special treatment required to model many 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. This shows 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 in Coq, with 983 lines of code for the consistency proof with β rules and a moderate increase to 1596 lines of code for the normalization proof with $\beta\eta$ rules. We discuss our choice of using Coq as our metatheory, including the use of off-the-shelf semantic engineering infrastructure and automation tools, in Section 7. Our proof development is available as supplementary material.
- We compare our work to existing proofs by logical relations and other proof techniques for proving consistency and normalization. We provide an overview of prior work (Section 9) and explain how various design decisions affect the size of our proof and its extensibility to additional features (Section 10).

The result of our work is an artifact targeted towards researchers familiar with the syntax of dependent types, as well as logical relations for simple or polymorphic types, and who wish to understand using logical relations *for* dependent types. The short mechanized proof is accompanied here by a pen-and-paper description using set-theoretic notation and terminology so that it is accessible to readers with a general mathematical background. This pen-and-paper proof purposefully follows the mechanized proof closely while avoiding Coq-specific details as much as possible, and lemmas are linked directly to their counterparts in the mechanization.

Not only does this close connection aid readers who wish to adopt proof assistants for mechanizing metatheory, this precision is also important for conveying the proof technique itself. Unlike properties derivable through purely syntactic means, proofs by logical relations make demands on the strength of the metatheory 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

$a, b, c, p, A, B, C ::=$	*Terms*
$\quad \mid \mathbf{Set}_i \mid x \mid \mathbf{Void} \mid \mathbf{absurd}\ b$	universes, variables, empty type, explosion
$\quad \mid \Pi x{:}A.B \mid \lambda x.a \mid a\ b$	function types, abstractions, applications
$\quad \mid a \sim b \in A \mid \mathbf{refl} \mid \mathbf{J}\ c\ p$	equality types, reflexivity proof, J eliminator
$\quad \mid \mathbf{Bool} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ a\ \mathbf{then}\ b_0\ \mathbf{else}\ b_1$	boolean type, true, false, conditionals
$\Gamma ::= \cdot \mid \Gamma, x : A$	*Typing contexts*
$\rho \in \mathit{Var} \to \mathit{Term}$	*Substitutions*

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. Its syntax is given in Figure 1. As a dependent type theory, terms and

types are collapsed into the same syntactic category. We use a Curry-style (extrinsic) syntax, where terms are not annotated by types, and only terms which participate in reduction are part of introduction and elimination forms.

The type $\mathbf{Set}_i$ represents a universe at level i, a natural number. 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. Type annotations in abstraction forms are omitted, and 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. Propositional equality types $a \sim b \in A$, represent an equality between terms a and b of the same type A, whose canonical inhabitant is the reflexive proof **refl**. Equality proofs p can be eliminated by the J eliminator $\mathbf{J}\, c\, p$, which casts the term c from one end of the equality to the other. Finally, we have booleans along with the standard boolean values and conditional expressions.

Our reduction and typing relations are defined in terms of *simultaneous substitutions* ρ, which are mappings from variables to terms. Simultaneous substitutions are more convenient to reason about than single substitutions, especially when dealing with semantic typing in Section 4. We use **id** as the identity substitution, and the extension operator $\rho[x \mapsto a]$ updates the substitution ρ to map the variable x to a rather than to $\rho(x)$. [SCW: Maybe we should handwave more here, and note that we aren't covering all of the details of the treatment of variable binding in the text. If readers want to understand that part, they should look at the Coq code, which uses de Bruijn indices anyways. Our goal is to convey the understanding of the Coq proof.] [YL: I agree. Should mention upfront that the presentation is not watertight and rigorous, though we have one in our mechanization] The substitution operation $a\{\rho\}$ replaces every free variable x of a by $\rho(x)$ simultaneously. A single substitution corresponds to composing extension and identity: $a\{b/x\} \coloneqq a\{\mathbf{id}[x \mapsto b]\}$.

2.1 Definitional equality via parallel reduction

Before we specify the typing rules, we first specify the equational theory used in the conversion rule T-Conv in Figure 3, known as *definitional equality* in dependent type theories because it defines the equivalence that the syntactic type system works up to. The definitional equality we use is *convertibility*: two terms are convertible if they reduce to a common form. The reduction that we use is *parallel reduction*, written $a \Rightarrow b$, with $a \Rightarrow^* b$ indicating its reflexive, transitive closure. The parallel reduction relation is defined in Figure 2, which omits reflexivity and congruence rules for brevity.

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 $a_1 \Rightarrow^* b$.

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

[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).

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

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

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

Fig. 2. Parallel reduction (β rules only)

Lemma 2.2 (Reflexivity (parallel reduction)[2])**.** *For all terms a, $a \Rightarrow a$.*

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

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

Lemma 2.5 (Diamond property[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 (Reflexivity (convertibility)[6])**.** *For all terms a, $a \Leftrightarrow a$.*

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

Lemma 2.8 (Transitivity (convertibility)[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 definitional equality 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 10.

Our definitional equality is untyped: the judgment 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 definitional equality takes the form $\Gamma \vdash a \equiv b : A$. By working with an untyped judgment, we can establish its properties independently of the type system and the logical relation, using well-established syntactic approaches. Siles and Herbelin (2012)

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

show the equivalence of Barendregt's Pure Type System, which employs untyped equality, and its variant that uses typed equality. This assures us that we do not lose generality working with a system with untyped conversion. [JC: I'm not too fond of this last sentence because we *do* lose the ability to add typed η equivalences.] [YL: removed the eta equivalence sentence because it might give the wrong message that we extended Siles' work with eta: Furthermore, we discuss how this definition can be extended with η-equivalence of functions in Section **??**] We compare this definition with type-directed approaches to equality in Section 10.

2.2 Syntactic Typing

Figure 3 gives the complete typing rules. The premises highlighted in gray can be shown to be admissible syntactically, though they are required to strengthen the inductive hypothesis of the fundamental theorem.

These rules 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. Rule T-Set ensures that each universe belongs to the next higher level. Rule T-Pi ensures predicative quantification by requiring that the domain and codomain types be typeable at the same universe level. Rule T-Abs ensures that all functions have well-formed dependent types. In the 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.

[SCW: Cassia did not follow our discussion of dependent pattern matching. We need to expand] The elimination form for booleans in rule T-If demonstrates dependent pattern matching: the type of the expression *depends* on the term being matched. This result type involves a *motive* A abstracted over a boolean. For the overall expression, x is filled in by the match target a, while in the **true** (resp. **false**) branch, x is filled in by **true** (resp. **false**). Informally, the type of each branch is more precise than those in a nondependent match expression, since it knows which branch it is in. An alternative perspective is that to prove $A\{a/x\}$, it suffices to show that it holds for the two canonical cases that a is **true** or is **false**.

Dependent pattern matching similarly occurs when eliminating equality types. Their well-formedness in rule T-Eq enforces that the endpoints of the equality have the same type, and there is a single canonical proof introduced by rule T-Refl. The elimination form in rule T-J takes a proof of an equality p between a and b, as well as an elimination body c. Here, the motive B is abstracted over the right endpoint x and an equality y between it and the fixed left endpoint a. The eliminator states that to prove $B\{b, p/x, y\}$, it suffices to prove the canonical case that p is **refl** (and consequently that b is a), which is witnessed by c.

The universe hierarchy and the boolean base type give the ability to compute a type using a term as input, referred to as *large elimination*. For example, the well-typed function $\lambda x.$**if** x **then Bool else Void** returns either **Bool** or **Void** depending on whether its input is **true** or **false**.

$\boxed{\vdash \Gamma}$ *(Context well-formedness)*

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

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

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

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

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

$$\textsc{T-Void}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{Void} : \mathbf{Set}_i} \qquad \textsc{T-Absurd}\ \frac{\Gamma \vdash b : \mathbf{Void} \qquad \Gamma \vdash A : \mathbf{Set}_i}{\Gamma \vdash \mathbf{absurd}\ b : A}$$

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

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

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

$$\textsc{T-Refl}\ \frac{\vdash \Gamma \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathbf{refl} : a \sim a \in A}$$

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

Fig. 3. Syntactic typing

3 Logical Relation

[SCW: We need to explicitly point out that the key ideas of this paper are discussed, here, in this section. We need to explicitly remark on why logical relations are difficult to define for dependent type theory and explain why this setting is more difficult than with simple types (STLC) or with polymorphic types (System F).

- Large eliminations
- Definitional equality (not all types look like types)

Should we be more explicit in our comparison with Girard's trick for polymorphic type? There, the definition stays recursive because it doesn't substitute for the variables in the function types. But that approach is not available in this setting, because not all quantified things are types. And we might need that information to interpret, say, equality types in the right way.] [SCW: We also need to explicitly point out that our logical relation is untyped. This has two benefits: it allows semantic typing to be meaningful independent from syntactic typing (cite Derek, forward reference to next section) and it avoids significant bookkeeping, especially in the case of Kripke logical relations (we need to define what these are). Is there a cost to an untyped relation?]

Our ultimate goal is to prove the following consistency property for our type theory.

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 a type with no constructors. A related property, 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 naïve attempt at proving Theorem 3.1 by induction on the derivation $\cdot \vdash a : \mathbf{Void}$ would succeed in 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. Unfortunately, there is nothing we know about b or a from the induction hypothesis because neither $\Pi x{:}A.B$ nor A are equal to **Void**, so 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.

[JC: Minor comment but ''proof by logical relation'' sounds grammatically strange to me, but I'm not sure whether ''proof by logical relations'' or ''proof by a logical relation'' would be any more correct...]

The challenge behind applying proofs by logical relation to dependent types stems from the difficulty in defining the logical relation itself. In simply typed languages, the logical relation is a recursive function over the type A. In dependently typed languages, the type A can take the form $(\lambda x.x)\ \mathbf{Bool}$, for example. 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. We later recover functionality of the relation in Lemma 3.8.

3.1 Definition of the Logical Relation

The logical relation[9] for λ^{Π}, which takes the form $[\![A]\!]^i_I \searrow S$, is defined as an inductively generated relation in Figure 4. The metavariables A and i are terms and naturals, respectively. The metavariables I and S are sets with the following signatures, using $\mathcal{P}(Term)$ to

[9] `semtyping.v:InterpExt`

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

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

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

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

Fig. 4. Logical relation for λ^{Π}

denote the powerset of the set of terms.

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

The function I is a family of sets of terms indexed by natural numbers strictly less than 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 predicativity of the system.

Predicativity allows us to tie the knot and obtain an interpretation for all universe levels as the judgment $[\![A]\!]^i \searrow S$, which says that the type A is a type at universe level i, *semantically* inhabited by terms from the set S.

Definition 3.2 (Logical relation for all universe levels)**.** $[\![A]\!]^i \searrow S$ is defined recursively by well-foundedness of the $<$ relation on natural numbers.

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

Our system is predicative because the interpretation of the ith universe is dependent only on universes strictly below i. This restriction ensures that the relation is well defined; otherwise, the definition of $[\![A]\!]^i \searrow S$ would not be well founded, and $[\![A]\!]^i_I \searrow S$ could 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 $[\![A]\!]^i_I \searrow S$ are admissible for $[\![A]\!]^i \searrow S$. For example, we can prove the following derived rules:

$$\text{IR-Void}\ \frac{}{[\![\mathbf{Void}]\!]^i \searrow \varnothing} \qquad \text{IR-Set}\ \frac{j < i}{[\![\mathbf{Set}_j]\!]^i \searrow \{A \mid \exists S, [\![A]\!]^j \searrow S\}}$$

In most informal presentations, instead of defining the logical relation in two steps as we have shown above, the rules for $[\![A]\!]^i \searrow S$ 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 $[\![A]\!]^i \searrow S$.

We next take a closer look at the remaining rules for the inductive relation $[\![A]\!]^i_I \searrow S$ 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. In particular, the **Void** type has no inhabitants, while 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 $\{a \mid a \Rightarrow^* \textbf{true} \vee a \Rightarrow^* \textbf{false}\}$ 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 in Section 5. [YL: Not sure what to cite from Derek Dreyer. I know his blog post about semantic type soundness but is there a good paper to cite? one of the rust papers?] [JC: Should there be an explanation of what Kripke-style logical relations are here?]

Rule I-Eq states 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. Recalling expression $(\lambda x.x)$ **Bool**, rule I-Red says that to know that $[\![(\lambda x.x)\ \textbf{Bool}]\!]^i_I \searrow S$ for some S, it suffices to show that $[\![\textbf{Bool}]\!]^i_I \searrow S$, since $(\lambda x.x)\ \textbf{Bool} \Rightarrow \textbf{Bool}$. The derivation for $[\![(\lambda x.x)\ \textbf{Bool}]\!]^i_I \searrow \{a \mid a \Rightarrow^* \textbf{true} \vee a \Rightarrow^* \textbf{false}\}$ therefore follows by composing rule I-Red and rule I-Bool.

Rule I-Pi is the most complex rule in our logical relation. It states that, to build the interpretation of a dependent function type, we first require the interpretation S of its domain A. Because the codomain depends on the function input, rather than a single interpretation, we require an interpretation $F(a)$ of the codomain $B\{a/x\}$ for each $a \in S$. The interpretation of the overall function type is the set of terms b such that for every term a in the interpretation S of A, $b\ a$ is in the interpretation $F(a)$ of $B\{a/x\}$.

However, this form of the rule is less convenient to work with, since it requires constructing the sets F separately from the proof that they are indeed interpretations of B. A more convenient form is the following rule I-PiAlt, which combines the existence of interpretations of the codomain with their proofs.

I-PiAlt

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

The second precondition now states that for every $a \in S$, there must exist some interpretation S_0 of $B\{a/x\}$. The interpretation of the overall function type then states that $b\ a$ must be in every interpretation S_0 of $B\{a/x\}$. Later, Lemma 3.8 proves functionality of the interpretation of types, which guarantees that there must only be one such interpretation. This alternate rule for the interpretation of function types follows from rule I-Pi.

Lemma 3.3 (I-PiAlt derivability[10])**.** *Rule* I-PiAlt *is derivable from rule* I-Pi.

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

While rule I-PiAlt is an instantiation of rule I-Pi, by functionality, the two rules are equivalent in the sense that every derivation involving rule I-Pi can be systematically replaced by rule I-PiAlt. Furthermore, functionality uniquely determines the function $F \in S \to \mathcal{P}(Term)$ 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 shown by Lemma 3.9, the corresponding inversion lemma for rule I-PiAlt.

Unfortunately, we cannot directly define the interpretation of function types by 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 on inductive definitions required by 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. [JC: What?] 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 postcondition as the relation grows, [JC: Why?] 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. Then by deriving rule I-PiAlt and its inversion lemma, we avoid needing to manipulate the function F directly.

3.2 Properties of the Logical Relation

In the rest of this section, we develop the theory of our logical relation with the goal of showing four key properties: irrelevance (Lemma 3.7), functionality (Lemma 3.8), cumulativity (Lemma 3.10), and the backward closure property (Lemma 3.13). These properties semantically correspond to ones that hold for syntactic typing, and allow us to prove that syntactic typing implies semantic typing, which we define in the next section.

In particular, irrelevance states that convertible types have the same interpretation, which is used to interpret rule T-Conv. Functionality with cumulativity shows that all interpretations of a type are the same across all universe levels, which allows us to treat a type that appears twice in a judgment uniformly, such as the function domain type in rule T-App. Finally, backward closure ensures that anything that reduces to a term in the interpretation

[10] `semtyping.v:InterpExt_Fun_nopf`

of a type is itself in the interpretation, which we use to handle β-reductions in the cases for introduction forms.

For the majority of the properties that we prove in this section, we need no information about the parameterized function I. Each property about $[\![A]\!]^i \searrow S$ follows as a corollary of a property about $[\![A]\!]^i_I \searrow S$ with no or few assumptions imposed on I. As a result, we state most of our lemmas in terms of $[\![A]\!]^i_I \searrow S$ without duplicating them in terms of $[\![A]\!]^i \searrow S$.

First, we prove inversion principles for our logical relation. Given $[\![A]\!]^i_I \searrow S$ where A is in some head form such as **Bool** or $\Pi x{:}A.B$, the inversion lemma allows us to say something about the set S. The proofs are simple, but we sketch out the case for functions to help readers confirm their understanding of rule I-Pi.

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

1. [11] *If* $[\![\mathbf{Void}]\!]^i_I \searrow S$, *then* $S = \varnothing$.
2. [12] *If* $[\![\mathbf{Bool}]\!]^i_I \searrow S$, *then* $S = \{a \mid a \Rightarrow^* \mathbf{true} \vee a \Rightarrow^* \mathbf{false}\}$.
3. [13] *If* $[\![a \sim b \in A]\!]^i_I \searrow S$, *then* $S = \{p \mid p \Rightarrow^* \mathbf{refl} \wedge a \Leftrightarrow b\}$.
4. [14] *If* $[\![\Pi x{:}A.B]\!]^i_I \searrow S_1$, *then there exist* $S \in \mathcal{P}(Term)$, $F \in S \to \mathcal{P}(Term)$ *such that:*
   - $[\![A]\!]^i_I \searrow S$
   - $\forall a$, *if* $a \in S$, *then* $[\![B\{a/x\}]\!]^i_I \searrow F(a)$
   - $S_1 = \{b \mid \forall a$, *if* $a \in S$, *then* $b\ a \in F(a)\}$
5. [15] *If* $[\![\mathbf{Set}_j]\!]^i_I \searrow S$, *then* $j < i$ *and* $S = I(j)$.

Proof We show only the inversion property for the function type. We start by induction on the derivation of $[\![\Pi x{:}A.B]\!]^i_I \searrow S_1$. There are only two possible cases we need to consider.

Rule I-Pi**:** Immediate.
Rule I-Red**:** We are given that $\Pi x{:}A.B \Rightarrow C$ and that $[\![C]\!]^i_I \searrow S_1$. By inversion on parallel reduction, we have that $C = \Pi x{:}A_0.B_0$ for some A_0, B_0, and that $A \Rightarrow A_0$ and $B \Rightarrow B_0$. From the induction hypothesis on $[\![\Pi x{:}A_0.B_0]\!]^i_I \searrow S_1$, there exist $S \in \mathcal{P}(Term)$ and $F \in S \to \mathcal{P}(Term)$ such that:

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

By Lemma 2.4, we have $B\{a/x\} \Rightarrow B_0\{a/x\}$ for all a. As a result, S and F satisfy the following properties by rule I-Red on $A \Rightarrow A_0$ and $B\{a/x\} \Rightarrow B_0\{a/x\}$, respectively:

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

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

An immediate consequence of the inversion principle for the interpretation of universes is that if a type is in such an interpretation, then that type itself has an interpretation.

[11] `semtyping.v:InterpExt_Void_inv`
[12] `semtyping.v:InterpExt_Bool_inv`
[13] `semtyping.v:InterpExt_Eq_inv`
[14] `semtyping.v:InterpExt_Fun_inv`
[15] `semtyping.v:InterpExt_Univ_inv`

Corollary 3.5.[16] *If* $[\![\mathbf{Set}_j]\!]^i \searrow S$ *and* $A \in S$*, then there exists some set* S_0 *such that* $[\![A]\!]^j \searrow S_0$ *and* $j < i$*.*

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 as well.

Lemma 3.6 (Forward preservation (l.r.)[17])**.** *If* $[\![A]\!]^i_I \searrow S$ *and* $A \Rightarrow B$*, then* $[\![B]\!]^i_I \searrow S$*.*

Proof We carry out the proof by induction on 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 $[\![B_1]\!]^i_I \searrow S$.

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

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

Corollary 3.7 (Irrelevance (l.r.)[18])**.** *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 that each type A interprets to a unique set S. The following lemma shows that our logical relation is indeed functional.

Lemma 3.8 (Functionality (l.r.)[19])**.** *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 on 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.

In rule I-Red, 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 forward preservation (Lemma 3.6) and $A \Rightarrow B$, we know that $[\![B]\!]^i_I \searrow S_1$. Then $S_0 = S_1$ immediately follows from the induction hypothesis. ■

Functionality allows us to prove an inversion lemma for the derivable rule I-PiAlt, which does not mention the function F found in rule I-Pi.

Lemma 3.9 (Alternate inversion of function types (l.r.)[20])**.** *If* $[\![\Pi x{:}A.B]\!]^i_I \searrow S$*, then there exists some* S_0 *such that:*

- $[\![A]\!]^i_I \searrow S_0$

[16] `semtyping.v:InterpUnivN_Univ_inv'`
[17] `semtyping.v:InterpExt_preservation`
[18] `semtyping.v:InterpUnivN_Coherent`
[19] `semtyping.v:InterpExt_deterministic`
[20] `semtyping.v:InterpExt_Fun_inv_nopf`

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

Proof Immediate from Lemmas 3.4 and 3.8. ■

The next lemma shows cumulativity of the logical relation: if a type has an interpretation at a lower universe level, then we obtain the same interpretation at a higher level.

Lemma 3.10 (Cumulativity (l.r.)[21]). *If* $[\![A]\!]^i_I \searrow S$ *and* $i < j$, *then* $[\![A]\!]^j_I \searrow S$.

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

Note that in the statement of cumulativity, we implicitly assume that I is defined on naturals strictly less than j. Then by cumulativity and trichotomy of the order on naturals (i.e. for any i, j, either $i < j$ or $i = j$ or $i > j$), we show that functionality holds even when considering interpretations of type at different universe levels.

Corollary 3.11 (Level-irrelevant functionality (l.r.)[22]). *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 3.8 and 3.10. ■

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 the actual instantiation of I to prove the rule I-Set case.

Definition 3.12 (Closure 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 $b \Rightarrow a$.

Lemma 3.13 (Backward closure of interpretations (l.r.)[23]). *If* $[\![A]\!]^i \searrow S$, *then* S *is closed under expansion.*

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

All cases are trivial except for the rule I-Set case, where the goal is to show that $I(j)$ is closed under expansion for all $j < i$; that is, if $B \Rightarrow A$ and $A \in I(j)$, then $B \in I(j)$. By the definition of I, this is equivalent to showing that if $B \Rightarrow A$ and $[\![A]\!]^j_I \searrow S$ for some S, then there exists some S_0 such that $[\![B]\!]^j_I \searrow S_0$. Letting S_0 be S, by rule I-Red, we have that $[\![B]\!]^j_I \searrow S$ as required. ■

4 Semantic Typing and Consistency

[SCW: Would it make sense to define the notation $a \in [\![A]\!]^i$ when there exists some S such that $[\![A]\!]^i \searrow S$ and $a \in S$?] [JC: I like this notation too, $a \in [\![A]\!]^i$]

[21] `semtyping.v:InterpExt_cumulative`
[22] `semtyping.v:InterpExt_deterministic'`
[23] `semtyping.v:InterpUnivN_back_clos`

In this section, we show that all closed, well-typed terms are contained in the interpretation of their types. In other words, $\cdot \vdash a : A$ implies that there exists a set S such that $[\![A]\!]^i \searrow S$ and $a \in S$ hold; we write $a \in [\![A]\!]^i$ as shorthand. This result gives us consistency because $[\![\mathbf{Void}]\!]^i \searrow \varnothing$ holds, so if there were a closed, well-typed term of type **Void**, it would be a member of the empty set, which is a contradiction.

To prove this result, we define a notion of semantic typing based on the logical relation and prove the fundamental lemma, which states that syntactic typing implies semantic typing. Semantic typing extends the 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 definitions on top of the logic relation is understood in the simply typed setting (Skorstengaard, 2019; Harper, 2022*a*; Pierce, 2002). In our setting, [JC: Is this trying to say that our setting is different from the simply typed setting, or the same as the simply typed setting? Are the definitions to follow on semantic typing taken from the works cited above, or are they our own modifications?] attempting to show that $\cdot \vdash a : A$ implies $a \in [\![A]\!]^i$ by induction on the derivation of $\cdot \vdash a : A$ will fail for rule T-Abs, where the induction hypothesis is not helpful since the body of the lambda term is typed under a nonempty context. Using the definition of semantic typing, we can state a strengthened property that is actually provable.

Definition 4.1 (Semantically well formed substitutions[24])**.** We say that a substitution ρ is semantically well formed with respect to the context Γ, written as $\rho \vDash \Gamma$, when

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

$\rho \vDash \Gamma$ states that for every variable x and its type A in the Γ, $\rho(x)$ is a term that inhabits every interpretation of $A\{\rho\}$. While functionality of the logical relation tells us each type has at most one interpretation, quantifying over all possible interpretations S makes the proofs slightly simpler. Since $\rho \vDash \Gamma$ typically appears as a hypothesis, S is easy to instantiate. The following structural properties handle the few cases where we need to prove $\rho \vDash \Gamma$ as a goal; the second property depends on functionality.

Lemma 4.2 (Well-formed ρ w.r.t. empty[25])**.** *$\rho \vDash \cdot$ holds for any ρ.*

Lemma 4.3 (Well-formed ρ w.r.t. cons[26])**.** *If $a \in [\![A]\!]^i$ and $\rho \vDash \Gamma$, then $\rho[x \mapsto a] \vDash \Gamma, x : A$.*

Semantic well-typedness is then defined using well formed substitutions to handle the contexts of open terms.

Definition 4.4 (Semantic typing[27])**.** We say that a is semantically typed as A under the context Γ, written $\Gamma \vDash a : A$, when

$$\forall \rho \vDash \Gamma, \exists j \text{ such that } a\{\rho\} \in [\![A\{\rho\}]\!]^j.$$

[24] `soundness.v:ρ_ok` [25] `soundness.v:ρ_ok_nil` [26] `soundness.v:ρ_ok_cons`
[27] `soundness.v:SemWt`

Semantic typing $\Gamma \vDash a : A$ says that for all well-formed substitutions $\rho \vDash \Gamma$, $a\{\rho\}$ is in the interpretation of $A\{\rho\}$ at some universe level. This definition is standard, though dependency of types on terms requires also applying the substitution ρ to the type A, and that $A\{\rho\}$ have an interpretation. Finally, we define semantic well-formedness of contexts, analogous to the relation $\vdash \Gamma$.

Definition 4.5 (Semantic context well-formedness[28])**.** We say that the context Γ is semantically well formed, written $\vDash \Gamma$, when

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

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

Lemma 4.6 (Empty context well-formedness[29])**.** $\vDash \cdot$ *holds.*

Lemma 4.7 (Cons context well-formedness[30])**.** *If* $\vDash \Gamma$ *and* $\Gamma \vDash A : \mathbf{Set}_i$*, then* $\vDash \Gamma, x : A$.

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

Lemma 4.8 (Set Inversion[31])**.** *The following are equivalent:*

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

Proof The forward direction is immediate by Lemma 3.5. We now consider the backward direction and show that $\Gamma \vDash A : \mathbf{Set}_i$ given the second bullet.

Suppose $\rho \vDash \Gamma$. 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; it is trivial to verify the conditions hold. ■

Next, we show some non-trivial cases of the fundamental theorem as top-level lemmas, namely for rule T-Var, rule T-Set, rule T-Pi, rule T-Abs, and rule T-App.

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 $[\![A\{\rho\}]\!]^i \searrow S$ and $\rho(x) \in S$ hold. By the definition of semantic context well-formedness, we deduce from $\vDash \Gamma$ and $x : A \in \Gamma$ that there exists some universe

[28] soundness.v:SemWff [29] soundness.v:SemWff_nil [30] soundness.v:SemWff_cons
[31] soundness.v:SemWt_Univ

level i such that $\Gamma \vDash A : \mathbf{Set}_i$. By the equivalence from Lemma 4.8, there exists an S such that $[\![A\{\rho\}]\!]^i \searrow S$. Finally, by the definition of $\rho \vDash \Gamma$, we know that $\rho(x) \in S$. ■

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 exist 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 3.9, the alternate inversion principle for rule I-Pi, there exists some S_0 such that the following hold:

- $[\![A\{\rho\}]\!]^i \searrow S_0$;
- $\forall a$, if $a \in S_0$, then $\exists S_1$, $[\![B\{\rho[x \mapsto a]\}]\!]^i \searrow S_1$; and
- $S = \{b \mid \forall a$, 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 $a \in S_0$ and $[\![B\{\rho[x \mapsto a]\}]\!]^i_I \searrow S_1$, we have $(\lambda x.b)\{\rho\}\,a \in S_1$. By Lemma 3.13, 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 $b\{\rho[x \mapsto a]\} \in S_1$. By $\Gamma, x : A \vDash b : B$, Lemma 4.3, and $a \in S_0$, we know that there exist some j, S_2 such that $[\![B\{\rho[x \mapsto a]\}]\!]^j \searrow S_2$ and $b\{\rho[x \mapsto a]\} \in S_2$. Finally, by level-irrelevant functionality of the logical relation (Lemma 3.11), we have that $S_1 = S_2$ and thus $b\{\rho[x \mapsto a]\} \in S_1$. ■

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 3.9, there exists some i and S_0 such that the following hold:

- $[\![A\{\rho\}]\!]^i \searrow S_0$;
- $\forall a_0$, if $a_0 \in S_0$, then $\exists S_1$, $[\![B\{\rho[x \mapsto a_0]\}]\!]^i \searrow S_1$; and
- $\forall a_0$, 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[32])**.**

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

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) are used to discharge their syntactic counterparts (e.g. Lemma 4.12 for rule T-Abs). The remaining cases not covered by those lemmas are similar to the ones already shown. ■

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

Proof Suppose $\cdot \vdash a : \mathbf{Void}$ is derivable. By the fundamental theorem, we have $\cdot \vDash a : \mathbf{Void}$, which states that for all $\rho \vDash \cdot, j, S$ such that $[\![\mathbf{Void}]\!]^j \searrow S$, we have $a\{\rho\} \in S$. By Lemma 4.2, any ρ we pick trivially satisfies $\rho \vDash \Gamma$; for convenience, we pick **id**, so that we have $a \in S$. However, by the **Void** case of the inversion property (Lemma 3.4), S must be the empty set, which is a contradiction. ■

The fundamental theorem also tells us that closed terms of type **Bool** reduce to either **true** or **false**.

Corollary 4.15 (Canonicity[33])**.** *If* $\cdot \vdash b : \mathbf{Bool}$, *then either* $b \Rightarrow^* \mathbf{true}$ *or* $b \Rightarrow^* \mathbf{false}$.

Proof Similar to above, using instead the **Bool** case of the inversion property. ■

5 Existence of β-normal forms

In this section, we extend the logical relation in Figure 4 to show the existence of β-normal forms for both open and closed well-typed terms. More precisely, we prove it possible to repeatedly apply parallel reduction to reduce a term to its unique normal form. Consequently, this shows that the convertibility relation is decidable.

This extension of the logical relation demonstrates that our proof technique can also be used to reason about the reduction properties of open terms, not just of terms after closing substitutions. Reasoning about open terms is especially important for dependently typed languages, as type checking involves working with open terms. [SCW: Add when we can find a reference: However, even non dependently-typed languages employ such techniques, especially in the case of relational semantics.] While the extension use well-known techniques, [JC: Does this need a citation?] it remains short and demonstrates the robustness of our initial framework.

Figure 5 gives the β-neutral forms e and β-normal forms f of λ^{Π}. Neutral terms consist of variables and elimination forms that do not reduce any further, while normal terms consist of neutral terms and other introduction forms. We use the judgment forms $\mathbf{ne}\ a$ and $\mathbf{nf}\ a$ to indicate that there exists a term e (resp. f) such that $a = e$ (resp. $a = f$). Additionally, we

[32] `soundness.v:soundness` [33] `soundness.v:canonicity`

$$e ::= x \mid e\,f \mid \mathbf{J}\,e\,f \mid \mathbf{if}\ e\ \mathbf{then}\ f\ \mathbf{else}\ f \mid \mathbf{absurd}\ e \qquad \textit{Neutral terms}$$
$$f ::= e \mid \mathbf{Set}_i \mid \Pi x{:}f.f \mid \mathbf{Bool} \mid f \sim f \in f \mid \mathbf{Void} \mid \lambda x.f \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{refl} \qquad \textit{Normal terms}$$

Fig. 5. β-neutral and β-normal forms

define predicates to describe terms that weak normalization: reduction to neutral or normal forms by parallel reduction.

Definition 5.1 (Weak normalization)**.** We define weak normalization of a term a to a neutral form, written **wne** a, or to a normal form, written **wn** a, as the following:

$$\mathbf{wne}\ a \iff \exists e, a \Rightarrow^* e$$
$$\mathbf{wn}\ a \iff \exists f, a \Rightarrow^* f$$

Weakly normalizing terms are composed of other weakly normalizing terms following the structure of neutral and normal forms. For example, an application normalizes to neutral form if the function normalizes to neutral form and the argument to normal form. Similar lemmas hold for function types[34], abstractions[35], conditionals[36], and absurdity[37].

Lemma 5.2 (Application (wne)[38])**.** *If* **wne** a *and* **wn** b*, then* **wne** $(a\ b)$.

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

The modifications to the logical relation[39] are given in Figure 6. The new rule I-Ne asserts that only terms that reduce to neutral forms inhabit the interpretation of neutral types. The remaining rules are updates to existing rules; we omit those for function types and universes as they are unchanged.

The changes in rule I-BoolNew and rule I-VoidNew follow the same pattern. An open term of type **Bool** may reduce to **true** or **false** as before, but may also reduce to a neutral term, such as a variable. Likewise, while in an empty context the interpretation of **Void** remains empty, if the context is not empty, then it may contain, for instance, a variable in the context of type **Void** or that can be eliminated into type **Void**.

In rule I-EqNew, we add preconditions asserting that a, b, and A are normal so that we only model normalizing equality types. Furthermore, we only require $a \Leftrightarrow b$ when the equality proof reduces to **refl**. If the proof term reduces to a neutral term, there is nothing we can assert about the relationship between a and b.

Because we are working with open terms, we need additional lemmas for parallel reduction. First, it preserves neutral and normal forms.

Lemma 5.3 (Preservation of neutral and normal forms (p.r.)[40])**.** *If* $a \Rightarrow b$*, then* **ne** a *implies* **ne** b*, and* **nf** a *implies* **nf** b.

[34] `normalform.v:wn_pi` [35] `normalform.v:wn_abs` [36] `normalform.v:wne_if`
[37] `normalform.v:wne_absurd` [38] `normalform.v:wne_app` [39] `semtypingopen.v:InterpExt`
[40] `normalform.v:nf_ne_preservation`

$\boxed{[\![A]\!]^i_I \searrow S}$ *(Extended logical relation)*

$$\textsc{I-Ne}\ \frac{\mathbf{ne}\ A}{[\![A]\!]^i_I \searrow \{a \mid \mathbf{wne}\ a\}} \qquad \textsc{I-VoidNew}\ \frac{}{[\![\mathbf{Void}]\!]^i_I \searrow \{a \mid \mathbf{wne}\ a\}}$$

$$\textsc{I-BoolNew}\ \frac{}{[\![\mathbf{Bool}]\!]^i_I \searrow \{a \mid a \Rightarrow^* \mathbf{true} \vee a \Rightarrow^* \mathbf{false} \vee \mathbf{wne}\ a\}}$$

$$\textsc{I-EqNew}\ \frac{\mathbf{nf}\ a \qquad \mathbf{nf}\ b \qquad \mathbf{nf}\ A}{[\![a \sim b \in A]\!]^i_I \searrow \{p \mid (p \Rightarrow^* \mathbf{refl} \wedge a \Leftrightarrow b) \vee \mathbf{wne}\ p\}}$$

Fig. 6. Extended logical relation for λ^{Π} (new and changed rules)

Since neutral and normal forms have no redexes, if a is neutral or normal, parallel reduction never β-reduces, so $a = b$ additionally holds, but preservation of neutral and normal forms is sufficient for our proof.

Next, parallel reduction also satisfies an antirenaming property. A *renaming* ξ is a special case of substitution where all variables are replaced by other variables. In this special case, if a renamed term reduces, then we can separate the reduction from the renaming, asserting that renaming never introduces new redexes.

Lemma 5.4 (Par antirenaming[41])**.** *If $a\{\xi\} \Rightarrow b_0$, then there exists some b such that $b\{\xi\} = b_0$ and $a \Rightarrow b$.*

[SCW: This lemma is only used to prove `wn_antirenaming`, which is then used more generally. Maybe we should replace it with that?] [YL: Anti-renaming is mentioned once in Section 6 when we talk about how confluence with η depends on anti-renaming for Par. I don't think that discussion in Section 6 is very important so I'm fine with replacing it]

Together, these lemmas prove an inversion property for weakly normalizing forms: if an application of a term to a variable is weakly normalizing, then the term must also be weakly normalizing.

Lemma 5.5.[42] *If* **wn** $(a\ x)$*, then* **wn** a*.*

Proof By induction over the length of the reduction sequence in **wn** $(a\ x)$. When the reduction is a β-reduction, a is a function and the application reduces to a renaming by x, so the conclusion follows from Lemma 5.4. ■

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 modifications to rules I-Void, I-Eq, and I-Bool can be discharged by Lemma 5.3.

[41] `normalform.v:Par_antirenaming` [42] `normalform.v:ext_wn`

Before we can prove the fundamental theorem and derive the normalization property as its corollary, we need to formulate and prove an *adequacy* property of the logical relation, which states that the interpretation of each type is a *reducibility candidate* (*CR*), which characterizes sets of weakly normalizing terms. In contrast to proving consistency, which only requires knowing that the interpretation of **Void** is empty, adequacy allows us to conclude that all terms in *every* interpretation have a normal form.

To prove adequacy, we need to strengthen it to with 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 define reducibility candidates as follows, inspired by Girard et al. (1989), but modified for weak normalization only. Adequacy then states that interpretations are in *CR*.

Definition 5.6 (Reducibility Candidates (*CR*)[43])**.** A set of terms S is in *CR* if and only if conditions CR_1 and CR_2 hold.

- $S \in CR_1 \iff \forall a$, if $\mathbf{wne}\ a$, then $a \in S$
- $S \in CR_2 \iff \forall a$, if $a \in S$, then $\mathbf{wn}\ a$

Lemma 5.7 (Adequacy)**.** [44] *If* $[\![A]\!]^i \searrow S$*, then* $S \in CR$*.*

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

By Definition 3.2, we have $[\![A]\!]^i_I \searrow S$, where $I(j) := \{A \mid \exists S, [\![A]\!]^j \searrow S\}$ for $j < i$. We then proceed by structural induction on the derivation of $[\![A]\!]^i_I \searrow S$. The interesting cases are rule I-Pi and rule I-Set. For rule I-Pi, proving $S \in CR_1$ requires Lemma 5.2. When proving $S \in CR_2$, by the structural induction hypotheses, we have that $\mathbf{wn}\ (b\ a)$ for any $\mathbf{wne}\ a$, while the goal is to show that $\mathbf{wn}\ b$. We pick an arbitrary x and conclude by applying Lemma 5.5 to $b\ x$.

The rule I-Set case is the most interesting. We must show that for all $j < i$, $\{A \mid \exists S, [\![A]\!]^j \searrow S\} \in CR$. We immediately know that $\{A \mid \exists S, [\![A]\!]^j \searrow S\} \in CR_1$ by rule I-Ne. It remains to show that $\{A \mid \exists S, [\![A]\!]^j \searrow S\} \in CR_2$, or equivalently that $[\![A]\!]^j_I \searrow S$ implies $\mathbf{wn}\ A$, where I is defined as above over $k < j$.

We assume $[\![A]\!]^j_I \searrow S$ and proceed once more by structural induction on its derivation. All cases are trivial except for rule I-Pi. The induction hypothesis immediately gives $\mathbf{wn}\ A$. To derive $\mathbf{wn}\ (\Pi x{:}A.B)$, it remains to show $\mathbf{wn}\ B$. We use the outermost induction hypothesis on $[\![A]\!]^j_I \searrow S$ to show that an arbitrary y semantically inhabits A by CR_1, followed by the structural induction hypothesis to derive $\mathbf{wn}\ (B\{y/x\})$. We conclude $\mathbf{wn}\ B$ by applying antirenaming (Lemma 5.4). ■

The formulation of semantic well-typedness and the fundamental lemma from Section 4 remains unchanged. The proof of the fundamental lemma[45] is still carried out by induction over the typing derivation, where the additional neutral term related cases are handled by adequacy. The normalization property then follows as a corollary of the fundamental theorem.

[43] `semtypingopen.v:CR` [44] `semtypingopen.v:adequacy` [45] `soundnessopen.v:soundness`

Corollary 5.8 (Existence of β-normal forms[46]). *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 $[\![A\{\rho\}]\!]^i \searrow S$ and $a\{\rho\} \in S$. We pick the ρ to be the identity substitution $\mathbf{id}$, and the condition $\mathbf{id} \vDash \Gamma$ is satisfied by adequacy, which says that variables, as neutral terms, semantically inhabit the interpretations of the types in Γ. Performing the identity substitutions, we then know that there exist i and S such that $[\![A]\!]^i \searrow S$ and $a \in S$. Finally, by adequacy again, 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, Abel et al. (2019) mechanize normalization generalized to open terms. In this setting, as above, adequacy must be proven before the fundamental theorem so that they can handle elimination rules such as rule T-App where the function is a neutral term. Dependent types make the adequacy proof 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 5.7).

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

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

[JC: Delete this section]

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.

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

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 5.4) 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.

[46] `soundnessopen.v:mltt_normalizing`

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 theorem shows us that every well-typed term has a β-normal form. However, in the presence of the η reduction rule, Lemma 5.3 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.[SCW: I tried to reword your sentence, but I am still not happy with it.]

7 Mechanization

To demonstrate the scale of our proof scripts, Figure 7 shows the number of non-blank, non-comment lines of code[47] for each file of our development, including the base consistency proof from Sections 3 and 4, along with the extension to β-normalization from Section 5. For comparison, we have also proven syntactic type safety through preservation [48] and progress[49].

[47] calculated by `tokei`: `https://github.com/XAMPPRocky/tokei`

[48] `syntactic_soundness.v`:`subject_reduction` [49] `syntactic_soundness.v`:`wt_progress`

	Consistency	Normalization	Safety
Libraries	15	=	=
Syntactic typing	87	=	=
Untyped reduction	345	=	=
Neutral/normal forms	—	287	—
Logical relation	295	385	—
Semantic soundness	195	215	—
Syntactic soundness	—	—	655
Total	934	1331	1099

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.

The Autosubst 2 tool takes our 16-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 autogenerated syntax file and other Autosubst library files (516 LOC total) are not included in the figure.

7.1 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 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 use this property in a few proofs, notably for inversion (Lemma 3.4) and functionality (Lemma 3.8), where we deal with equalities of the interpretations, which are sets of terms.

In the Coq mechanization, sets of terms $\mathcal{P}(Term)$ are encoded as the predicates over terms `tm -> Prop` asserting whether the term is in the set. We want to be able to show that two predicates P and Q are equal when $\forall x, P(x) \iff Q(x)$ holds, but this predicate extensionality property does not hold in axiom-free Coq, and requires both functional and propositional extensionality.

7.2 Encoding the logical relation

As discussed, the logical relations $[\![A]\!]^i_I \searrow S$ in Figures 4 and 6 are encoded as inductive definitions indexed by sets of terms representing the semantic inhabitants of A. S therefore has type `tm -> Prop`, while I has type `forall j, j < i -> tm -> Prop`, representing a set of types in universes strictly smaller than i. Because the overall inductive definition represents a functional relation between a type at a universe level and its interpretation as a set of terms, it too lives in `Prop`.

Due to the limitations of eliminating inductives in `Prop`, the encoding of the rule for function types differs from the presentation in rule I-Pi, reproduced below.

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

While the first and last premises are propositions, S and F represent sets of terms, which are not propositions. While this is not an issue when proving properties of the logical relation, which never require projecting out these sets, it is an issue when proving the fundamental theorem, and in particular proving the function case in Lemma 4.11. By the definition of semantic typing, the induction hypothesis (after some instantiation) gives

$$\forall a \in S, \exists i, S_0 \text{ such that } [\![B\{a/x\}]\!]^i \searrow S_0 \text{ and } b\, a \in S_0.$$

We would like to define F as the function that projects out S_1 from the induction hypothesis, but the existential quantification being a proposition prevents us from doing so without the axiom of choice. To avoid relying on classical reasoning, we instead encode F as a functional relation $R \in S \times \mathcal{P}(Term)$ such that $\forall a, \exists S_0, (a, S_0) \in R$ holds. The actual encoding is given as rule I-PiCoq below.

$$\text{I-PiCoq}\quad \frac{[\![A]\!]_I^i \searrow S \qquad R \in S \times \mathcal{P}(Term) \qquad \forall a, \exists S_0, (a, S_0) \in R \qquad \forall a, \forall S_0, \text{if } (a, S_0) \in R, \text{ then } [\![B\{a/x\}]\!]_I^i \searrow S_0}{[\![\Pi x{:}A.B]\!]_I^i \searrow \{b \mid \forall a, \forall S_0, \text{if } (a, S_0) \in R, \text{ then } b\, a \in S_0\}}$$

From this rule, we are still able to derive rule I-PiAlt as before. The benefit of working with the derived rule is that it does not involve R, which is uniquely determined to be $\{(a, S_0) \mid \text{if } a \in S, \text{ then } [\![B\{a/x\}]\!]_I^i \searrow S_0\}$ as before.

An alternative to these various encodings in Coq is to use *induction–recursion* (IR) (Dybjer, 2000), which is available in Agda (Agda Development Team, 2023). IR permits mutually defining an inductive definition along with a recursive function on the inductive. We can change our perspective on the logical relation and view it as two separate pieces: an inductive interpretation $[\![A]\!]^i$ of types, and a recursive interpretation $a \in [\![A]\!]^i$ of types as sets of terms. The rule for function types and its interpretation would be defined as follows.

$$\text{IIR-Pi}\quad \frac{[\![A]\!]^i \qquad \forall a, \text{if } a \in [\![A]\!]^i, \text{ then } [\![B\{a/x\}]\!]^i}{[\![\Pi x{:}A.B]\!]^i}$$

$$b \in [\![\Pi x{:}A.B]\!]^i \coloneqq \forall a, \text{if } a \in [\![A]\!]^i, \text{ then } b\, a \in [\![B\{a/x\}]\!]^i$$

Because the inductive definition no longer needs to quantify over all sets of terms, we can work entirely within Agda's lowest `Set` universe without worrying about impredicativity or universe polymorphism. However, to define the above, we still need an intermediate definition involving the interpretation I of types at $j < i$ and tie the knot as before. While the same properties are provable, the inductive–recursive logical relation changes the structure and the order of these proofs; for instance, a weaker form of functionality is provable before and is required to prove forward preservation.

Further details of the IR encoding can be found in our Agda mechanization of the consistency of λ^{Π}. Due to the lack of automation support, proofs about syntactic properties of terms and proofs using well-founded induction on levels are written manually. Even so, the total number of nonblack, noncomment lines of code remains small at below 1300 lines, and functional extensionality is still the only axiom used.

7.3 Automation in Coq

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 `asimpl` tactic 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, which is never specified 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 only use lemmas or invert derivations specified by the `use` or `inv` flags.

[JC: Just a digression but I find that CoqHammer automated proofs are *not* good documentation compared to how we might have otherwise written the proofs manually. While

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

$$\textsc{T-Nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{Nat} : \mathbf{Set}_i} \qquad \textsc{T-Zero}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{zero} : \mathbf{Nat}} \qquad \textsc{T-Suc}\ \frac{\Gamma \vdash a : \mathbf{Nat}}{\Gamma \vdash \mathbf{succ}\ a : \mathbf{Nat}}$$

$$\textsc{T-Ind}\ \frac{\Gamma \vdash a : A\{\mathbf{zero}/x\} \qquad \Gamma, x : \mathbf{Nat} \vdash A : \mathbf{Set}_i \qquad \Gamma, x : \mathbf{Nat}, y : A \vdash b : A\{\mathbf{succ}\ x/x\} \qquad \Gamma \vdash c : \mathbf{Nat}}{\Gamma \vdash \mathbf{ind}\ a(xy.b)c : A\{c/x\}}$$

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

$$\textsc{I-Nat}\ \frac{}{[\![\mathbf{Nat}]\!]^i_I \searrow \{a \mid a \Rightarrow^* b \wedge \vdash_{\mathbb{N}} b\}}$$

$\boxed{\vdash_{\mathbb{N}} a}$ *(Natural values)*

$$\textsc{N-Zero}\ \frac{}{\vdash_{\mathbb{N}} \mathbf{zero}} \qquad \textsc{N-Neu}\ \frac{\mathbf{ne}\ a}{\vdash_{\mathbb{N}} a} \qquad \textsc{N-Suc}\ \frac{\vdash_{\mathbb{N}} a}{\vdash_{\mathbb{N}} \mathbf{succ}\ a}$$

Fig. 8. Rules for natural numbers

the flags indicate what lemmas and inversions *may* be used, the `best` tactic does not delete unused ones, so there's no way of knowing which ones actually *are* used, and the automation hides away *how* they are used. Automation makes proofs resilient to change when things go right, but when things go wrong, it becomes much harder to discern what lemmas I need to add when I don't even know where the existing ones are used.]

8 Extensions

In this section, we discuss how the normalization proof of λ^{Π} can be extended to incorporate features including natural numbers, dependent pairs, cumulative universes with subtyping, and the η law for functions. These extensions have been implemented in our development. More advanced features such as type-directed η laws are out of the scope of this paper, and we instead point the readers to existing work for techniques that can be readily adapted to our minimal development.

[JC: We may want to briefly mention the LOC for each addition?]

8.1 Natural Numbers

We can replace the boolean type with the natural number type as our observable type. By introducing an infinite data type to λ^{Π}, we significantly increase its expressiveness.

The syntactic and semantic rules for natural numbers can be found in Figure 8. T-Zero and T-Suc allow us to construct natural numbers. T-Ind allows us to eliminate natural numbers through the induction principle where a is the base case, b the inductive case, and c the natural number being eliminated.

Rule I-Nat gives us the semantic interpretation of natural number: a term a semantically inhabits **Nat** if a reduces to some term b such that $\vdash_{\mathbb{N}} b$ holds. The $\vdash_{\mathbb{N}} b$ judgment inductively characterizes the normal form of (potentially open) terms of type **Nat** and is particularly useful in the T-Ind case of the fundamental theorem. Let $\rho \vDash \Gamma$, the induction hypothesis $\Gamma \vDash c : \mathbf{Nat}$ tells us that $c\{\rho\} \Rightarrow^* b$ for $\vdash_{\mathbb{N}} b$. The proof then follows by induction over the derivation of $\vdash_{\mathbb{N}} b$. When b is neutral, we conclude the proof with Adequacy. When b is **zero**, we conclude the proof with the outer induction hypothesis $\Gamma \vDash a : A\{\mathbf{zero}/x\}$. When b is a successor, we conclude the proof with the outer induction hypothesis $\Gamma, x : \mathbf{Nat}, y : A \vDash b : A\{\mathbf{succ}\ x/x\}$.

8.2 Cumulativity

The semantic model we build in Section 3 satisfies Cumulativity (l.r.)[50].

8.3 $\beta\eta$-normal forms

9 Related Work

9.1 Logical relations for dependent types

In the most general sense, a logical relation is a practical technique that uses a type-indexed relation to strengthen the induction hypothesis for the property of interest. This technique originates from Tait (1967), who maps types to sets of terms satisfying certain properties related to reduction. The same idea is explained by 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 notation. 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 function that takes a type and returns a set; however, in constructive type theory, the metalogic of Coq and Agda, the return type of the interpretation function depends on the derivation of the well-typedness of its input. The body of the interpretation function uses the well-typedness derivation, along with a proof classifying whether the input is a kind, a type, or a term, to determine whether the argument of an application should be erased during interpretation.

As a result, Geuvers' interpretation causes difficulties for modern proof assistants. Due to the impredicativity of the object language, the proof cannot be encoded in Agda, whose metatheory is predicative. However, even though Coq supports impredicativity,

[50] `semtyping.v:InterpExt_cumulative`

the proof relevance of the well-typedness derivations would require juggling between the impredicative but irrelevant `Prop` sort and the predicative but relevant `Type` sort.

More recent work such by Abel and Scherer (2012) and Abel et al. (2008) makes the definitions more explicit and precise and thus more directly encodable in proof assistants. Unlike λ^{Π}, their systems support type-directed η laws. These type-directed equational rules complicate the proof significantly since the confluence of full reduction is not available until after the fundamental theorem is proven. Therefore, instead of a logical predicate, they need a relational model with syntactic type annotations to remove the early reliance on confluence, in spite of the object type theory being intensional.

Coquand (2019, 2023) uses a *proof-relevant* logical relation to prove canonicity and normalization for intensional dependent type theories. In contrast to the proof-irrelevant logical predicate we present, Coquand's proof does not rely on the confluence of reduction, and can be easily extended to more η laws without relying on a complex relational model. Instead of working with concrete syntax, Coquand works directly with an algebraic representation of dependent type theory using categories with families (CwF) (Hofmann, 1997), though the idea of proof-relevant logical relations for dependent types can be expressed independently of category theory (Barras, 2012).

More abstractly, Coquand's proof technique is an instance of a categorical construction called Artin gluing (Kaposi et al., 2019*a*). Different metatheoretic properties, such as normalization, parametricity, and Π-injectivity, can be obtained by gluing the initial CwF with the corresponding CwF model. Sterling (2021) introduces the notion of synthetic Tait computability, which allows further simplification of such proofs by working in the internal language formed by gluing. However, proof-relevant logical relations are more demanding on the expressive power of the metatheory. For example, the mechanization of NbE by Altenkirch and Kaposi (2016) relies on quotient inductive–inductive types (QIIT) (Kaposi et al., 2019*b*). [JC: I don't understand how the QIIT model relates to gluing, or what using QIIT has to do with proof relevance of the logical relation.]

So far, the works mentioned interpret types as sets or relations on deep representations of terms. In contrast, shallow embeddings interpret types as sets of *meta-level* terms in some type theory. The logical relation interprets the object language into the meta language, and its soundness holds by virtue of the interpretation function being well typed and definable. Examples include work by McBride (2010), who mixes deep and shallow representations, and work by Kaposi et al. (2019*c*), who directly implement the signature of a CwF as Agda terms. The key property of shallow embeddings is that object-level conversion is represented as meta-level equality, which has the benefit of delegating handling conversion to the type checker, but with the drawback that object conversion must coincide with meta equality. Conversion cannot be customized to convert more terms than dictated by their interpretations.

9.2 Mechanized logical relations for dependent types

[SCW: Extra columns in Fig 10: Predicate vs. Relational interpretation (we can explain this) / Typed vs. Untyped interpretation]

Figure 9 presents several mechanized proofs that feature logical relations for dependently typed languages. Each of these proofs is significantly larger than our development, but they

	U	Ind	C	LE	A	Main results	LOC
λ^{Π} (this work)	$\mathbb{N}$	Eq, Bool	U	✓	1	Consistency, normalization	1331
λ^{θ}	0	Eq, Nat	U	✗	1	Consistency	~8k
Core Nuprl	$\mathbb{N}$	W types	E	✓	2	Consistency	~330k
NBE-in-Coq	1	Nat	T	✗	2	Correctness of NbE	~20k
$\lambda^{\Pi U \mathbb{N}}$	1	Nat, Σ	T	✓	2	Decidability: conversion	~10k
MLTT-à-la-Coq	1	Eq, Nat, Σ	T	✓	2	Decidibility: type checking	~30k

Universes:	Countable ($\mathbb{N}$), zero (0), one (1)
Inductives:	Equality types (Eq), naturals (Nat), dependent pairs (Σ), W types
Conversion:	Untyped (U), typed (T), extensional (E)
Large Elimination:	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-à-la-Coq	Adjedj et al. (2024)

Fig. 9. Feature matrix for dependently typed languages with mechanized logical relations

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. For a basic comparison of development size, we include the approximate nonblank, noncomment lines of proof code for each project.

Casinghino et al. (2014) introduce λ^{θ}, a dependently typed programming language that uses modalities to distinguish logical proofs from programs. The consistency proof of λ^{θ}'s logical fragment has been mechanized in Coq using 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).

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 induction–recursion, which is available in Agda but not in Coq. Adjedj et al. (2024) adapts the logical relation by Abel et al. (2017) to the predicative fragment of Coq and further extends the decidability of type conversion result to the decidability of type checking a bidirectional type system.

Anand and Rahli (2014) mechanize the metatheory of Nuprl (Constable et al., 1986) in Coq. The object theory is an extensional type theory with dependent functions, inductive

types, partial types, and a full universe hierarchy. They construct a PER model in Coq to show the logical consistency of the language. Their development has been further extended with intersection types, union types, and quotient types. Wieczorek and Biernacki (2018) mechanize a 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, large elimination is not supported. Both Anand and Rahli (2014) and Wieczorek and Biernacki (2018) leverage Coq's impredicative `Prop` sort to define the interpretation of dependent function types and thus are closely related to our mechanization. Anand and Rahli (2014) further show it possible to encode a finite universe hierarchy using neither impredicativity nor induction–recursion. Their encoding of a countable universe hierarchy relies on impredicativity, similar to our development. [JC: Wait I'm confused, do they use impredicativity or not?]

9.3 Other mechanized metatheory of dependent types

Barras (2010) and Wang and Barras (2013) assign set-theoretic semantics to dependent type theory in Coq. Unlike the previous efforts, which focus on predicative type theory and direct reducibility models, they tackle extensions of CC^{ω}, which extends the Calculus of Constructions with predicative universes on top of the impredicative sort. We choose to focus on a syntactic term model to avoid 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 Calculus of Cumulative Inductive Constructions (pCuIC), Coq's core calculus, assuming strong normalization. Weirich et al. (2017) define System D, a core calculus of dependent Haskell, and prove the syntactic type safety of the type system. Because System D includes nontermination, they prove consistency of definitional equality from the confluence of parallel reduction. [JC: Why specifically mention System D? Surely there are other more notable (in reviewers' eyes) dependent languages for which type safety have been proven?]

Compared to the systems described here, the most notable feature we are missing is impredicativity, which is known to be difficult to model when the impredicative sort is at the bottom of a predicative universe hierarchy. [JC: This makes it sound like it's easy when it's at the top of a predicative universe hierarchy, which I don't think is what you meant; did you mean when impredicativity is combined with some sort of large elimination of base types/ type-level computation?] 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.

10 Discussion

[YL: I'm not sure how strong of a statement we can make about our proof technique. We've already demonstrated how to address type-level dependency when defining a logical relation through a simple example. Claiming that our proof structure is better seems quite ambitious, but I think there's a middle point where we claim that adding moderate features like typed reduction doesn't instantly make our code size expand all the way from 1000 to 20,000 without saying the other developments are just verbose for no good reason] [SCW: I think we can find reasons for much of the differences. Am I missing any? While it would be difficult to assign numbers to each of the deltas, I think it is believable that when put together they add up to a lot.

- We don't include inductive or coinductive datatypes. We don't include cumulativity. We don't include Prop. We don't include universe polymorphism.
- We state our equality algorithmically instead of declaratively. On one hand, this gives us automatic inversion principles when working with definition. Furthermore, we don't need to prove the equivalence between an algorithmic version and a declarative specification.
- Our equality requires a simple decision algorithm and isn't type directed.
- We don't prove decidability of type checking. (And, it is not provable for our system, because we lack type annotations on functions. We should point this out.)
- Our logical relation is unary and untyped. The latter means that we don't require the bookkeeping of a Kripke logical relation when reasoning about open terms. I don't know why unary relations are shorter.
- CoqHammer leads to short proofs.

] [YL: Just one more technical point to add, though it's in the text already: the logical relation is closed backward by full reduction rather than weak-head/deterministic reduction. This requires an early confluence result to show that the logrel is deterministic/functional but simplifies everything else (e.g. conversion is justified immediately by our preservation theorem, but that is not the case if you use weak head reduction).

Also, regarding the first point, cumulativity only exists in Barras's work. Inductive, (maybe coinductive?), can be found in nuprl, metacoq, and maybe Barras's work.

The 20,000 - 30,000 LoC mechanization are all about small languages with pretty much the same features as our language except for your second and third bullet point. martin-lof a la coq, Abel's work, and nbe in coq aren't that richer in feature otherwise. None includes cumulativity (they only have one predicative universe)

The 400,000 NuPRL in Coq probably falls into a different category because they are trying to mechanize a full practical language] [SCW: The Coq-Coq-Correct paper (extended version) includes a (predicative) universe hierarchy, universe polymorphism, inductive/coinductive types. But they don't show consistency. Their development is 300k LOC.]

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. [SCW: I'm getting confused by this paragraph. Does this reorganization sense: Our language is simpler than Nuprl, because it doesn't have extensional equality. It is simpler than Agda, because it doesn't have type-directed equality. Both of these cases require the definition of a binary logical relation, that defines a notion of semantic equality between terms. This relation justifies the injectivity of Π types and justify the validity of η-conversion among other properties.] [SCW: Furthermore our proof is also simpler because we don't need prove the correctness of the NBE algorithm, which is used to show the decidability of Agda's type-directed equivalence. Therefore, we don't need to define this algorithm and show that it is sound and complete with respect to the type-directed equality. Instead, to show the decidability of our untyped equivalence, we need only show that terms have $\beta\eta$ normal forms.] [YL: Makes sense. Though Abel's work doesn't use nbe but a recursive binary algorithm. Rewrote the paragraph above and commented out the original]

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. [SCW: Need to define Kripke-style. Also the other proofs need Kripke style because they are defining typed relations, not untyped relations.] [YL: I wonder if we can just assume some more technical knowledge from the readers in this

section.] 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.[SCW: confluence depends on Pi injectivity? I thought it was only needed for subject reduction][YL: it's transitive. Confluence depends on subject reduction, which in turn depends on pi injectivity. Maybe it's worth spelling out the details] 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.

11 Conclusion

In this work, we present a short and mechanized proof by logical relations for a dependently typed language with a predicative universe hierarchy, large elimination, a propositional equality type, and dependent eliminators. From the fundamental theorem for the logical relation, we prove consistency and canonicity, then show the extensibility of our approach by proving the existence of $\beta(\eta)$-normal forms and adding natural numbers and cumulativity to the language. These extensions only require 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

Agda Development Team. 2023. *Agda*. https://wiki.portal.chalmers.se/agda/Main/HomePage

Thorsten Altenkirch and Ambrus Kaposi. 2016. Normalisation by Evaluation for Dependent Types. (2016), 16 pages. https://doi.org/10.4230/LIPICS.FSCD.2016.6 Artwork Size: 16 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany.

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.

Bruno Barras. 2012. Semantical investigations in intuitionistic set theory and type theories with inductive families. *Habilitation, Université Paris* 7 (2012).

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. 2019. Canonicity and normalization for dependent type theory. *Theoretical Computer Science* 777 (July 2019), 184–191. https://doi.org/10.1016/j.tcs.2019.01.015

Thierry Coquand. 2023. Reduction Free Normalisation for a proof irrelevant type of propositions. *Logical Methods in Computer Science* Volume 19, Issue 3 (July 2023), 8818. https://doi.org/10.46298/lmcs-19(3:5)2023

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.

Peter Dybjer. 2000. A general formulation of simultaneous inductive-recursive definitions in type theory. *The Journal of Symbolic Logic* 65, 2 (June 2000), 525–549. https://doi.org/10.2307/2586554

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.

Martin Hofmann. 1997. *Syntax and Semantics of Dependent Types*. Cambridge University Press, 79–130.

Ambrus Kaposi, Simon Huber, and Christian Sattler. 2019a. Gluing for Type Theory. *LIPIcs, Volume 131, FSCD 2019* 131 (2019), 25:1–25:19. https://doi.org/10.4230/LIPICS.FSCD.2019.25 Artwork Size: 19 pages, 515812 bytes ISBN: 9783959771078 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik Version Number: 1.0.

Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. 2019b. Constructing quotient inductive-inductive types. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–24. https://doi.org/10.1145/3290315

Ambrus Kaposi, András Kovács, and Nicolai Kraus. 2019c. Shallow Embedding of Type Theory is Morally Correct. In *Mathematics of Program Construction*, Graham Hutton (Ed.). Vol. 11825. Springer International Publishing, 329–365. https://doi.org/10.1007/978-3-030-33636-3_12 Series Title: Lecture Notes in Computer Science.

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

Conor McBride. 2010. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN Workshop on Generic Programming* (Baltimore, Maryland, USA) *(WGP '10)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1863495.1863497

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

Jonathan Sterling. 2021. First Steps in Synthetic Tait Computability. (2021).

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