Combining Proofs and Programs in a Dependently Typed Language

Chris Casinghino  Vilhelm Sjöberg  Stephanie Weirich
University of Pennsylvania
{ccasin,vilhelm,sweirich}@cis.upenn.edu

Abstract

Most dependently-typed programming languages either require that all expressions terminate (e.g. Coq, Agda, and Epigram), or allow infinite loops but are inconsistent when viewed as logics (e.g. Haskell, ATS, Ωmega). Here, we combine these two approaches into a single dependently-typed core language. The language is composed of two fragments that share a common syntax and overlapping semantics: a logic that guarantees total correctness, and a call-by-value programming language that guarantees type safety but not termination. The two fragments may interact: logical expressions may be used as programs; the logic may soundly reason about potentially nonterminating programs; programs can require logical proofs as arguments; and “mobile” program values, including proofs computed at runtime, may be used as evidence by the logic. This language allows programmers to work with total and partial functions uniformly, providing a smooth path from functional programming to dependently-typed programming.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory

Keywords  Dependent types; Termination; General recursion

1. Introduction

Dependently typed languages have developed along two different traditions, distinguished by their attitude towards nonterminating programs. On the one hand, languages like Cayenne [6], ATS [13], Ωmega [35] and Haskell [29] treat dependently-typed programming as an extension of ordinary functional programming. These languages enhance ordinary functional programs, defined by general recursion, with more expressive types. On the other hand, languages like Coq [40], Agda [28] and Epigram [23] treat dependently typed programming as a use-case of constructive theorem proving. These systems disallow nontermination because an infinite loop can be given any type and would therefore make the logic inconsistent.

We would like balance between proving and programming, with neither activity given preferential treatment. Although we are sympathetic to the ideal that all programs should be proven correct, we understand that there are practical reasons not to do so. Instead, we desire a language for heterogeneous verification, allowing programmers to devote their verification budget to critical sections. Such a language must support general recursion as natively as a functional programming language, yet at the same time must provide the expressive reasoning capabilities of a constructive logic proof assistant.

In support of this goal, we propose a novel language that is composed of two fragments: a logical fragment where every expression is known to terminate, and a programmatic fragment that does not provide this assurance. The key idea of our work is to distinguish between these fragments by indexing the typing judgement with a consistency classifier θ that may be L ("logic") or P ("program"), thus

\[ \Gamma \vdash_\theta a : A \]

When θ is L, the Curry-Howard Isomorphism applies, and we may consider a a proof of the theorem A. When θ is P, the only interpretation of a is as a functional program. Making this distinction means that one language can subsume both functional programming and constructive logic by embedding each in their respective fragments. However, these activities are not too far apart—the syntax and semantics of the two fragments overlap considerably, because the distinction between them is made through typing.

In this paper, we explore the consequences of this design in the context of a dependently-typed programming language, focusing on the following mechanisms that foster interaction between the two fragments.

- First, we define the logical language as a sublanguage of the programmatic language, so that all logical expressions can be used as programs. (Of course, the programmatic language includes forms that are not available to the logic, including general recursion and the elimination of iso-recursive types.)
- We allow uniform reasoning for logical and programmatic expressions through a heterogeneous equality type. Two expressions can be shown to be equal based on their evaluation, which is the same for both fragments. Equality proofs can be used implicitly by the type system.
- We internalize the labeled typing judgment as a new type form Aθ. This type can be used by either fragment to manipulate values belonging to the other.
- We identify a set of “mobile types”—those whose values can freely move between the fragments.

To demonstrate the soundness and consistency of these mechanisms, we define a core dependently-typed language, called \( \lambda^\theta \), that supports these interactions (see Sections 2 and 3). In addition to the Aθ type, this language includes dependent functions, products, propositional equality, natural numbers, sums, recursive functions.
and iso-recursive types. We prove that this language is type safe and that the L fragment is normalizing and logically consistent (Section 4). Our normalization proof uses a combination of traditional and step-indexed logical relations. All of our metatheoretic results have been completely machine-checked using the Coq proof assistant and are available online.\footnote{Proofs available at \url{http://www.cis.upenn.edu/~ccasin/papers/combinng-coq.tgz}}

We also explore how our ideas interact with other programming language features. We have implemented a prototype language, Zombie, based on the semantics of λ₀, and discuss that implementation in Section 5. Zombie extends λ₀ with features that are convenient for dependently-typed programming: parametric polymorphism, type-level computation, user-defined datatypes, and implicit arguments. We have developed a number of examples using Zombie; the implementation is available online.\footnote{Implementation available at \url{https://code.google.com/p/trellys} in the branch branches/zombie-trellys-POPL14/}

We are not the first to consider the combination of total and partial programming in the setting of dependently-typed languages. Partial types \footnote{1} and the conductive partiality monad \footnote{2} embed general recursive programs into constructive logic by modeling nontermination. Alternatively, languages such as Idris \footnote{10}, Aura \footnote{18}, and F* \footnote{38} identify a restricted sublanguage of pure total functions. However, neither of these approaches provide equal support for total and partial programming. We compare them to our work in Section 6.

### 1.1 Combining Proofs and Programs

Before explaining the semantics of λ₀, we conclude this section with a number of examples to demonstrate the key ideas.

In Zombie, declarations must indicate whether they belong to the logical or programmatic fragment of the language. For example, a boolean negation operation is trivially terminating, so it is checkable in the logical fragment, as indicated by the tag \logic {log} in its definition:

\[
\mathtt{log}\ \mathtt{not} : \mathtt{Bool} \to \mathtt{Bool} \\
\mathtt{not} \ b = \mathtt{if} \ b \ \mathtt{then} \ \mathtt{False} \ \mathtt{else} \ \mathtt{True}
\]

Likewise, addition for natural numbers can be shown terminating via natural number induction. In the case expression below, plus may be called on any subterm of its argument. The argument \mathtt{n\_eq} is a proof that \mathtt{n} is a subterm of \mathtt{n}:

\[
\mathtt{log}\ \mathtt{plus} : \mathtt{Nat} \to \mathtt{Nat} \to \mathtt{Nat} \\
\mathtt{plus} \ n \ m = \\
\textit{case} \ n \ [n\_eq] \ of \\
\mathtt{Zero} \to m \\
\mathtt{Succ} \ n' \to \mathtt{Succ} \ (\mathtt{plus} \ n' \ n\_eq \ m)
\]

Alternatively, the following natural number division function diverges when \mathtt{m} is 0, so it must be tagged with \prog. The \rec keyword indicates that this function is implemented using general recursion.

\[
\mathtt{prog}\ \mathtt{div} : \mathtt{Nat} \to \mathtt{Nat} \to \mathtt{Nat} \\
\mathtt{rec} \ \mathtt{div} \ n \ m = \\
\textit{if} \ \mathtt{lt} \ n \ m \ \mathtt{then} \ \mathtt{0} \\
\textit{else} \ \mathtt{plus} \ \mathtt{1} \ (\mathtt{div} \ (\mathtt{minus} \ n \ m) \ m)
\]

### Subsumption

All proofs can be used as programs. In the above example, even though the plus operation is logical, we can seamlessly use it (and other logical operations such as \mathtt{lt} and \mathtt{minus}) directly in a programmatic term, and call it on an argument whose termination behavior is unknown. Thus, the fact that we know that plus terminates does not restrict how it may be used—we do not need to duplicate its definition for it to be available to both fragments.

#### Proofs containing programs

The \@\-type allows values to be embedded from one fragment into another. For example, the logical language can safely manipulate programmatic values as long as their types indicate (with \@P) that they are programmatic. Below, consider the definition of a Maybe datatype that could contain arbitrary programs.

\[
\mathtt{data}\ \mathtt{Maybe}\ (A : \mathtt{Type})\ \mathtt{where} \\
\mathtt{Nothing} \\
\mathtt{Just}\ \mathtt{of}\ (A @ P)
\]

As long as the programmatic component is treated carefully, expressions in the logical fragment can work with this data structure. This includes constructing values of the Maybe type, and pattern matching on the data structure.

\[
\mathtt{log}\ \mathtt{md3} : \mathtt{Maybe}\ (\mathtt{Nat} \to \mathtt{Nat}) \\
\mathtt{md3} = \mathtt{Just} \ \langle x. \ \mathtt{div} \ 3 \ x \rangle
\]

\[
\mathtt{log}\ \mathtt{foo} : \mathtt{Maybe}\ (\mathtt{Nat} \to \mathtt{Nat}) \to \mathtt{Nat} \to \mathtt{Maybe}\ \mathtt{Nat} \\
\mathtt{foo} \ x = \mathtt{case} \ x \ \mathtt{of} \\
\mathtt{Just} \ f \to \mathtt{Just} \ (f \ y) \\
\mathtt{Nothing} \to \mathtt{Nothing}
\]

However, if the programmatic component is ever used, then the definition must be marked as programmatic, as an embedded function could cause divergence.

\[
\mathtt{prog}\ \mathtt{bar} : \mathtt{Maybe}\ (\mathtt{Nat} \to \mathtt{Nat}) \to \mathtt{Nat} \to \mathtt{Maybe}\ \mathtt{Nat} \\
\mathtt{bar} \ x \ y = \mathtt{case} \ x \ \mathtt{of} \\
\mathtt{Just} \ f \to \mathtt{Just} \ (f \ y) \\
\mathtt{Nothing} \to \mathtt{Nothing}
\]

#### Proofs about programs

Having defined the programmatic function \mathtt{div}, we might wish to verify facts about it. As a simple example, we prove that \mathtt{div} \ 6 \ 3 evaluates to 2. We can state and prove these facts using the logical language, even though the object of study may not terminate.

\[
\mathtt{log}\ \mathtt{div63} : \mathtt{div} \ 6 \ 3 = 2 \\
\mathtt{div63} = \textit{refl}
\]

The proof above (\textit{refl}) is valid when both sides of an equality proposition evaluate to the same result. (To avoid infinite loops, the typechecker will give up and signal an error if the expression does not reach a normal form within 1000 steps. If more evaluation is required the programmer can write e.g. \textit{refl 5000}). In languages like Aura or F*, this theorem cannot even be stated because non-value expressions such as \mathtt{div} \ 6 \ 3 cannot appear in types. This example illustrates an important property of our language, which we call \textit{freedom of speech}: although proofs cannot themselves use general recursion, they are allowed to \textit{refer} to arbitrary programmatic expressions.

As a more complicated example, we might wish to prove that if the divisor is not zero, then the result is less than the dividend. In other words:

\[
\mathtt{log}\ \mathtt{div}\_\mathtt{le} : (n : \mathtt{Nat}) \to (m : \mathtt{Nat}) \to (\mathtt{eq} \ m \ \mathtt{0} = \mathtt{False}) \to (\mathtt{le} \ (\mathtt{div} \ m \ n) \ n = \mathtt{True})
\]

Above, \mathtt{eq} is an equality function for natural numbers and \mathtt{le} \ m \ n determines whether \mathtt{m} \leq \mathtt{n}. We do not show proof of the above theorem here, though it is available with our implementation. The
proof itself uses strong natural number induction to simultaneously show both that division terminates and that the property above holds for the result.

Note that we can only show properties that are provable via finite reduction sequences. For example, we cannot show that division diverges when the dividend is 0, because that divergence is not finitely observable. (The logic does not have a general principle for reasoning about nonterminating programs, such as fixed-point induction. We return to this issue in Section 6.)

Programs that return proofs. An alternative to writing separate proofs about nonterminating programs is to give the programs themselves more specific types that express their correctness. For example, consider writing a SAT solver that we do not want to prove terminating.

A SAT solver takes a formula of $n$ variables and, if the formula is satisfiable, returns a satisfying assignment for some subset of those variables. We can represent the result of a SAT solver using the following datatype declaration. The result for a given formula is either an assignment together with a proof that that assignment satisfies the formula, or UNSAT when the formula is unsatisfiable.

\[
\text{data Ans} \ (n : \text{Nat}) \ (\text{form} : \text{Formula} \ n) : \text{Type where}
\]

- \[\text{SAT} \ (\text{assign} : \text{Vector} \ (\text{Maybe} \ \text{Bool}) \ n) \ (\text{proof} : \text{satisfies assign form} = \text{(Just True : Maybe Bool)}) \]

\[\text{UNSAT}\]

The main loop of the solver itself takes a formula and the current assignment and returns whether that assignment can be extended to a satisfying one. If the current assignment is known to be satisfying, then that one is returned. \text{Zombie} can automatically fill in the \_ below with the proof that the assignment satisfies the formula. If the assignment is known to invalidate the formula, then the result is UNSAT. Otherwise the algorithm must search for an extension to the assignment using techniques such as unit propagation, pure literal assignment, or merely trying both possibilities for an unassigned variable.

\[
\text{prog solver} : (n : \text{Nat}) \rightarrow (\text{formula} : \text{Formula} \ n) \\
\rightarrow \text{Vector} \ (\text{Maybe} \ \text{Bool}) \ n \\
\rightarrow \text{Ans} \ n \ \text{formula} \ @ \ L
\]

\[\text{solver n formula assign} =\]

- \[\text{case} \ (\text{satisfies assign formula}) \ of\]
  - \[\text{Just True} \rightarrow \text{SAT assign } _\_\]
  - \[\text{Just False} \rightarrow \text{UNSAT}\]
  - \[\text{Nothing} \rightarrow _\_\_\_
\]

Since the solver is written in the programmatic fragment, it may not terminate. It also may fail to find an assignment even though the formula was satisfiable. However, the type of this function is more informative than if it had been written in ML or Haskell. The \_L in its type indicates that if it \textit{does} return a proof of satisfiability, then that value was type checked in the logical fragment.

When a program contains subexpressions from both fragments, values can be handled more freely than expressions. For example, a logical expression cannot call solver directly because of the possibility of divergence. However, if the result of that call has already been bound to a variable, then the logic has access to that result.

\[
\text{let prog f} = (\ldots : \text{Formula} \ n) \ \text{in}
\]

\[
\text{let log empty} = \text{repeat} \ (\text{Nothing} : \text{Maybe} \ \text{Bool}) \ n \ \text{in}
\]

\[
\text{let prog isSat} = (\text{solver n empty : Ans} \ n \ @ \text{L}) \ \text{in}
\]

\[
\text{let log prf} = \text{case isSat of}
\]

\[\text{SAT assignment prf} \rightarrow\]

- \[\ldots \ \text{use proof of satisfiability} \ldots\]

\[\text{UNSAT} \rightarrow \ldots\]

\[\theta \quad a, b, A, B \quad ::= \ L \mid P \mid a = b \mid \text{Nat} \ [A + B] \mid \Sigma a : A.B \mid \mu x : A \ A@\theta \mid x \ [\lambda x : A. B] \mid \text{rec f} \ x \ a \mid \text{ind f} \ x \ a \mid \text{refl} \ [\text{inl} \ a] \mid \text{inr} \ b \mid \text{scase} \ a \ \text{of} \ {\text{inl} \ x \Rightarrow a_1; \text{inr} \ y \Rightarrow a_2} \\
\text{case} \ a \ \text{of} \ {\text{inl} \ x \Rightarrow \text{prf}} \mid \text{inr} \ b \\
\text{Z} \ [S a] \mid \text{ncase} \ a \ \text{of} \ {Z \Rightarrow a_1; S \Rightarrow a_2} \\
\text{roll} \ a \mid \text{unroll} \ a
\]

\[
\begin{align*}
\theta & \ ::= \ L \mid P \\
a, b, A, B & \ ::= \ * \mid (x : A) \rightarrow B \mid a = b \\
\text{Nat} & \ ::= A + B \mid \Sigma a : A.B \mid \mu x : A \ A@\theta \\
x & \ ::= \lambda x : A. B \mid \text{rec f} \ x \ a \mid \text{ind f} \ x \ a \mid \text{refl} \\
\text{inl} & \ ::= \text{inl} \ a \mid \text{inr} \ b \mid \text{scase} \ a \ \text{of} \ {\text{inl} \ x \Rightarrow a_1; \text{inr} \ y \Rightarrow a_2} \\
\text{Z} & \ ::= S a \mid \text{ncase} \ a \ \text{of} \ {Z \Rightarrow a_1; S \Rightarrow a_2} \\
\text{roll} & \ ::= \text{roll} \ a \mid \text{unroll} \ a
\end{align*}
\]

Figures 1. Expressions, values, and operational semantics (excerpt)

Mobile types Finally, some types have the same meaning in both fragments, so they do not benefit from being tagged with a consistency classifier. For example, a value of type \text{Nat} can never cause divergence, so it is safe to be used in logical expressions even when not marked as \_L. Similarly, the \text{Ans} type above is also mobile, so the \_L annotation on the type of \text{solver} is actually unnecessary. This observation simplifies programming as the only function arguments that must be annotated with their fragment are those that are not mobile.

2. The $\lambda^\theta$ language

We begin our technical development with an overview of the formal language, $\lambda^\theta$. This language is based on a call-by-value (CBV) variant of lambda calculus. Its syntax is shown in Figure 1 for uniformity, terms, types and the single kind $*$ (the “type” of types) are drawn from the same syntactic category, as in pure type systems [7]. The first two lines of the figure list the type forms, the following lines list the terms. By convention, we use lowercase metavariables $a, b$ for expressions that are terms and uppercase metavariables $A, B$ for expressions that are types.

The $\lambda^\theta$ values $v$ and key rules of the operational semantics are also shown in Figure 1. The reduction relation $a \rightsquigarrow b$ defines a small-step call-by-value semantics. The slightly unusual beta rule for natural number induction ($\text{SI}_\text{ND}$) is described in Section 2.1.
The starting point for \( \lambda \) and programmatic fragments of the language. In the next section, we introduce the novel features of the language, including its basic judgements (Section 2.1), and treatment of equality (Section 2.2). We do not want terms with the same runtime behavior to be considered unequal just because they have different annotations. Due to the lack of annotations, it is not possible to algorithmically compute the type of a \( \lambda \) term. This is not a problem because CBV evaluation only substitutes values for variables and eliminated by \( \texttt{pcase} \) and \( \texttt{unroll} \). Values include the standard components of functional programming: recursive functions \( \texttt{rec} \) \( f \), \( a \), nonrecursive functions \( \lambda x.a \), natural numbers (constructed by \( \texttt{Z} \) and \( \texttt{S} \) \( a \) and eliminated by \( \texttt{case} \)), disjoint unions (constructed by \( \texttt{inl} a \) and \( \texttt{inr} a \) and eliminated by \( \texttt{case} \)), and recursive data (introduced by \( \texttt{roll} a \) and eliminated by \( \texttt{unroll} a \)). Values also include \( \star \), all type forms, a trivial equality proof \( \texttt{refl} \), and \( \texttt{trivial equality proof} \).

Due to the lack of annotations, it is not possible to algorithmically compute the type of a \( \lambda \) term. This is not a problem because we do not want terms with the same runtime behavior to be considered unequal just because they have different annotations.

The rest of this section describes the specific details of \( \lambda \), including its basic judgements (Section 2.1), and treatment of equality (Section 2.2). In the next section, we introduce the novel features of our language that permit the interaction between the logical and programmatic fragments of the language.

### 2.1 Classifying terminating and nonterminating expressions

The starting point for \( \lambda \) is a dependent type theory where the typing judgment \( \Gamma \vdash a : A \) is indexed by a consistency classifier \( \theta \). The judgement is designed so that expressions that type check at \( \Gamma \) always terminate.

Figure 2 shows the typing rules for the basic building blocks of the language—variables, functions and various data structures and their types. Because we work with a collapsed syntax, we use the type system to identify which expressions are types: \( A \) is a well-formed type if \( \Gamma \vdash A : \star \).

Contexts are lists of assumptions about the types of variables.

\[
\Gamma ::= \emptyset \mid \Gamma, x : \theta A
\]

Each variable in the context is tagged with \( \theta \) to indicate its fragment, and this tag is checked in the TVAR typing rule. A context is valid, written \( \vdash \Gamma \), if each type \( A \) is valid in the corresponding fragment.

The rules TARR, TSigma, TSum, and TMU check types for well-foundedness. For example, \( \text{TARR} \) checks a function type by checking the \( \text{the domain and range} \). We discuss the premise \( \text{Mobile}(A) \), which asserts that \( A \) is a mobile type, in Section 3.2.

There are three ways to define functions in \( \lambda \). Rule TLAM types non-recursive \( \lambda \)-expressions in the logical fragment, whereas rule TREC types general recursive \( \lambda \)-expressions and can only be used in the programmatic fragment.

Additionally, terminating recursion over natural numbers is provided in the logical fragment by rule TIND. When typechecking the body of a terminating recursive function \( \text{ind} f x b \), the recursive

---


**Figure 2. Typing: variables, functions, and datatypes (rules for Nat omitted)**
call $f$ takes an extra argument proving that it is being applied to the predecessor of the initial argument $x$. This ensures termination. When beta-reducing such an expression, this argument is ignored by wrapping the function in an extra lambda (rule $\text{SINo}$ from Figure 1).

The rule for function application, $\text{TAPP}$, differs from the usual application rule in pure dependently-typed languages in the additional third premise $\Gamma \vdash_\theta \theta_1/a \mid x : B \mid s$, which checks that the result type is well-formed. Some rules of the language (such as $\beta$-reduction) are sensitive to whether terms are values. Because values include variables, substituting an expression $a$ for a value $x$ could cause $B$ to no longer type check.

Any dependently typed language that combines pure and effectful code will likely have to restrict the application rule in some way. Previous work [18, 21, 38] uses a more restrictive typing for applications, by splitting it into two rules: one which permits only value dependency and requires the argument to be a value, and one which allows a non-dependent function to be applied to an arbitrary argument. Since substituting a value can never violate a value restriction in $B$, our application rule subsumes the value-dependent version. Likewise, in the case of no dependency, the premise can never fail because the substitution has no effect on $B$.

Being able to call dependent functions with non-value arguments is useful when writing explicit proofs. For example, a programmer may want to first prove a lemma about addition

$$\log \text{plus_zero} : (n : \text{Nat}) \rightarrow \text{plus} n 0 = n$$

and then instantiate the lemma to prove a theorem about a particular expression in the logical fragment.

$$\text{plus_zero} \ f \ x \ : \ \text{plus} \ (f \ x) \ 0 = (f \ x)$$

The rules for sum types ($\text{TSUM}$, $\text{TINL}$, $\text{TINR}$, and $\text{TSCASE}$) provide dependent case analysis. The term $\text{case}$ binds the logical variable $z$ inside both branches of the case. This variable provides an equality between the scrutinee and the pattern of the branch so that type checking is flow-sensitive. At runtime, this variable is replaced by refl because the scrutinee must match the pattern for the branch to be taken.

The rules for dependent products ($\text{TSIGMA}$, $\text{TPAIR}$, $\text{TPCASE}$) allow the type of the second component of the pair to depend on the value of the first component. As with function application, the premise $\Gamma \vdash_\theta \theta_1/a \mid z : B \mid s$ ensures that substituting the expression $a$ does not violate any assumptions made about the value $z$ in the type of the second component. Analogously to sums, the eliminator for pairs makes available a logical proof $z$ that equates the scrutinee to the pattern in the body of the match. The availability of this equality means that the strong elimination forms (projections) for $\Sigma$-types are derivable.

Finally, the rules $\text{TMU}$, $\text{TROLL}$ and $\text{TUNROLL}$ deal with general recursive types. These are the standard rules for iso-recursive types (see, e.g., [30]). But recursive types with negative occurrences—that is, with the recursive variable appearing to the left of an arrow, such as $\mu x. (x \rightarrow \text{Nat})$—are a potential source of nontermination. To ensure normalization, it suffices to restrict the elimination rule $\text{TUNROLL}$ to be in $\mathcal{P}$. The introduction rule $\text{TROLL}$ can be used in both fragments. This reflects the fact that it is not dangerous to construct negative datatype values; the potential nontermination comes from their elimination.

### 2.2 Reasoning about equivalence

A big benefit of combining termination-checking with dependent types is that it is possible to write proofs about programs. For example, in the introduction we showed a proof that when the divisor is not zero, natural number division produces a result less than the dividend. Our rules for propositional equality (Figure 3) are designed to support such reasoning uniformly, based only on the runtime behavior of the expressions being equated, and independent of the fragment that they are defined in.

Therefore, the rule $\text{TEQ}$ shows that the type $a = b$ is well-formed and in the logical fragment even when $a$ and $b$ can be type checked only programmatically. This is freedom of speech: proofs can refer to nonterminating programs.

The term $\text{refl}$ is the primitive proof of equality. Rule $\text{TREFL}$ says that $\text{refl}$ is a proof of $a = b$ just when $a$ and $b$ reduce to a common expression. The notion of reduction used in the rule is parallel reduction, denoted $a \Rightarrow b$. This relation extends the ordinary evaluation $a \rightarrow b$ by allowing reduction under binders, e.g. $(\lambda x. 1 + 1) \Rightarrow (\lambda x. 2)$ even though $(\lambda x. 1 + 1)$ is already a value. Having this extra flexibility makes equality more expressive and simplifies the proof of preservation.

Proven equalities are used to substitute expressions in types by the elimination rule $\text{TCONV}$. The proof term is checked in $\mathcal{L}$ to ensure it is a valid proof. We demand that the equality proof used in conversion type checks in the logical fragment for type safety. All types are inhabited in the programmatic fragment, so if we permitted the user to convert using a programmatic proof of, say, $\text{Nat} = \text{Nat} \rightarrow \text{Nat}$, it would be easy to create a stuck term. Similar to $\text{TAPP}$, we need to check that $b_2$ does not violate any value restrictions, so the last premise checks the well-formedness of the type given to the converted term. Rule $\text{TCONV}$ is quite general, and may be used to change some small part of $A$ or the entire type by picking $x$ for $A$.

This treatment of equality is a variant of Sjöberg et al. [34]. However, that setting did not include a logical sublanguage; instead it enforced soundness by requiring the proof term used in conversion to be a value.

Uses of $\text{TCONV}$ are not marked in the term because they are not relevant at runtime. Again, types should describe terms without interfering with equality; we do not want terms with the same runtime behavior to be considered unequal due to uses of conversion.

### 3. Interactions between the fragments

What is interesting about $\lambda^\theta$ is how its two fragments interact. In the introduction, we discussed ways in which logical and programmatic terms work together. Below, we discuss the technical machinery of the type system that supports this interaction.

#### 3.1 Subsumption

Every logical expression can be safely used programatically. We reflect this fact into the type system by the rule $\text{TSUB}$, which says that if a term $a$ type checks logically, then it will also type check programatically. For example, a logical term can always be supplied to a function expecting a programmatic argument. This rule is useful to avoid code duplication. A function defined in the
Logical fragment can be used without penalty in the programmatic fragment.

Subsumption also eliminates duplication in the design of the language. For example, we need only one type $a = b$ to talk about when two programmatic or two logical terms are equal. In fact, we can also equate logical and programmatic expressions.

### 3.2 Internalized Consistency Classification

To provide a general mechanism for logical expressions to appear in programs and programmatic values to appear in proofs, we introduce a type that internalizes the typing judgment, written $A@\theta$. Nonterminating programs can take logical proofs as preconditions (with functions of type $(x : A@\theta) \rightarrow B$), return them as post-conditions (with functions of type $(x : A) \rightarrow (B@\theta)$), and store them in data structures (with pairs of type $\Sigma x : A. (B@\theta)$). At the same time, logical lemmas can use $\theta$ to manipulate values from the programmatic fragment.

The rules for the $A@\theta$ type appear in Figure 4. Intuitively, the judgment $\Gamma \vdash a : A@\theta$ holds if the fragment $\theta$ is not marked in the syntax, so the reduction behavior of an expression is unaffected by whether the type system deems it to be provably terminating or not.

For example, a recursive function $f$ can require an argument to be a proof by marking it $0@\theta$, e.g., $A@\theta \rightarrow B$, forcing that argument to be checked in fragment $L$. Similarly, a logical lemma $g$ can be applied to a programmatic value by marking it $@P$:

$$
\begin{array}{c}
\Gamma \vdash a : A@L \quad \Gamma \vdash f : A@L \rightarrow B \\
\Gamma \vdash a : A@L \\
\Gamma \vdash f : A@L \\
\Gamma \vdash g : A@P \rightarrow B \\
\Gamma \vdash g : A@P \\
\end{array}
$$

Of course, $g$ can only be defined in the logical fragment if it is careful to not use its argument in unsafe ways. For example, using TCONV we can prove a lemma of type:

$$
(n : \text{Nat}) \rightarrow (f : (\text{Nat} \rightarrow \text{Nat}@P) \rightarrow (f \text{ (plus } n \theta) = f \ n))
$$

because reasoning about $f$ does not require calling $f$ at runtime.

### 3.3 Mobile types

The consistency classifier tracks which expressions are known to come from a normalizing language. For some types of values, however, the rules described so far can be unnecessarily conservative. For example, while a programmatic expression of type $\text{Nat}$ may diverge, a programmatic value of that type is just a number, so we can treat it as if it were logical. On the other hand, we can not treat

---

4 This is one drawback of working in a strict rather than a lazy language. If we know that $f$ is nonstrict, then this application is indeed safe.
a programmatic function value as logical, since it might cause non-termination when applied.

The rule TMVAL (Figure 5) allows values to be moved from the programmatic to the logical fragment. It relies on an auxiliary judgment Mobile(\(A\)). Intuitively, a type is mobile if the same set of values inhabit the type when \(\theta = L\) and when \(\theta = P\). In particular, these types do not include functions (though any type may be made mobile by tagging its fragment with \(\emptyset\)).

Concretely, the natural number type Nat is mobile, as is the primitive equality type (which is inhabited by the single constructor refl, as discussed in Section 2.2). Any \(@\)-type is mobile, since it fixes a particular \(\theta\) independent of the one on the typing judgment. Sum and pair types are mobile if their component types are.

Even if a sum type is not mobile, it is always safe to do one level of pattern matching on one of its values, since such a value must start with a constructor. We reflect that in the rule TSCASE\(^*\), which generalizes TSCASE from the previous section. This rule allows a scrutinee that type checks in one fragment \(\theta\) to be eliminated in another fragment \(\theta'\). This lets the logical language reason by case analysis on programmatic values. Similarly, TPPCASE\(^*\) is a more general version of the rule TPCASE. The two rules shown here are the ones actually included in our formalization.

The mobile rule lets the programmer write simpler types, because mobile types never need to be tagged with logical classifiers. For example, without loss of generality we can give a function the type \((a = b) \to B\) instead of \(((a = b)@L) \to B\), since when needed, the body of the function can treat the argument as logical through TMVAL. Similarly, multiple \(\emptyset\)'s have no effect beyond the innermost \(\emptyset\) in a type. Values of type \(A@P@L@P@L@P\) can be used as if they had type \(A@P\).

In fact, the arguments to functions must always have mobile types. This restriction, enforced by rule TARR, means that higher-order functions must use \(\emptyset\)-types to specify which fragment their arguments belong to. For example, the type \((\text{Nat} \to \text{Nat}) \to A\) is not well-formed, so the programmer must choose either \((\text{Nat} \to \text{Nat})@L@L\) \(\to A\) or \((\text{Nat} \to \text{Nat})@P\) \(\to A\).

In either case, programmers benefit from implicit unboxing. For example, checking well-formedness of a type like

\[(f : (\text{Nat} \to \text{Nat}@P)) \to f \text{ plus n \emptyset} = f \text{ n}\]

implicitly uses TUNBOXVAL. But the equation still talks about the expression \(f\ n\). If we instead had to use explicit unboxing to eliminate the \(\emptyset\)-type, as in \(\text{unbox } f\ n\), there would be no way to write a logical lemma proving the original equation. By contrast, mobile arguments do not need nor benefit from tagging.

The reason that function arguments must be mobile is to account for contravariance. Through subsumption, we can introduce a function in the logical fragment and use it in the programmatic:

\[
\frac{
\Gamma, x : A \vdash b : B
}{
\Gamma \vdash (\lambda x. b) : (x : A) \to B}
\]

TLMAM

\[
\frac{
\Gamma \vdash P (\lambda x. b) : (x : A) \to B
}{
\Gamma \vdash P}
\]

TTSUB

Here, the definition of \(b\) assumed \(x\) was logical, yet when the function is called it can be given a programmatic argument. For this derivation to be sound, we need to know that \(A\) means the same thing in the two fragments, which is exactly what Mobile(\(A\)) checks.

4. Metatheory

We now describe the metatheory of \(\lambda^L\). We are interested in two properties. First, that the entire language is type safe, including both the L and P fragments. Second, that any closed term in the L fragment normalizes, which implies logical consistency.

Type safety is proven using standard progress and preservation theorems. Since the rules TCONV and TCONTRA allow stuck terms to type check given a contradiction, the progress theorem depends on logical consistency. For this reason, we first prove preservation, then normalization and consistency, and finally progress.

The theorems in this paper have been checked in Coq. To prove certain facts about our logical relation we needed a standard axiom of functional extensionality. This axiom is known to be consistent with Coq’s logic [41].
inversion we know either $\Gamma \vdash \alpha : A$.

\[
\begin{array}{rcl}
\Gamma \vdash \alpha : \; \Gamma' & \vdash \alpha : \ (x : A_1 \to A_2) & \Rightarrow \ (x : B_1 \to B_2) \\
\Gamma \vdash \alpha : A_1 = B_1 & {\rm TARRINV1} & \Gamma \vdash \alpha : A_1 = B_1 \\
\Gamma \vdash \alpha : \; \Gamma' \vdash \alpha : \; \Gamma' \vdash \alpha : \; \Gamma' \vdash \alpha : \ B & {\rm TCONTRA} & \Gamma \vdash \alpha : \ B
\end{array}
\]

Figure 6. Typing: discrimination and injectivity of type constructors (injectivity rules for $\otimes$-, $\mu$-, pair- and sum-types omitted).

4.2 Normalization and Progress

Our proof of normalization builds upon the standard Girard-Tait reducibility method \cite{Girard05} in a CBV-style formulation. The crux of this method is to define a “type interpretation”. For each type $A$ we define a set of values $V^\rho_A$ that check in fragment $\theta$ (the additional inputs $\rho$ and $k$ are discussed below). The definition of the type interpretation (Figure 7) is a logical relation and follows the structure of $A$.

Our main theorem is that the interpretation is “sound”: any closed logical expression $a$ of type $A$ reduces to a value in $V^\rho_A$. The rules $\mathsf{TUNBOXVAL}$ and $\mathsf{TMVAL}$ can move values from $P$ to $\mathsf{L}$, so for the proof to go through we must generalize the soundness theorem to also characterize expressions in $P$. For these values we prove a partial correctness property: if a closed programmatic expression $a$ of type $A$ reduces to a value, then the value is in $V^\rho_A$. These invariants are summarized by a computational type interpretation $V^\rho_A$, which identifies sets of (non-value) expressions, and is defined mutually with $V^\rho_A$.

The type interpretation for programmatic expressions must account for recursive functions and recursive types, which means that it cannot be defined by recursion on $A$. Instead, we use step-indexing \cite{Bjorner:01}. The interpretation is indexed by a number $k$. Any value $v$ in $V^\rho_A$ will be “well-behaved” for at least $k$ steps of execution. The interpretation is defined by well-founded recursion on the lexicographically ordered triple $(k, A, \mathcal{I})$, where $\mathcal{I}$ is one of $\mathcal{C}$ or $\mathcal{V}$ with $\mathcal{V} \subset \mathcal{C}$.

However, the usual formulation of a step-indexed type interpretation only lends itself to proving safety properties—it tells us that an expression will not do anything bad for the next $k$ steps. By contrast, normalization is a liveness property: every expression will eventually do something good (namely reduce to a value). In our proof, we use a hybrid approach by only counting steps that happen in the $P$ fragment. The difference can be seen by comparing the definitions of $V^\rho_A$ and $V^\rho_B$. which say “$j \leq k'$ and “$j < k'$" respectively. If all $\theta$s in a derivation are $\mathsf{L}$, then no inequalities are strict, so the step-count $k$ never needs to decrease.

The input $\rho$ is a substitution mapping free variables of $A$ to values. We use $\rho$ when interpreting equality types. The type $a_1 = a_2$ is interpreted as the singleton set $\{\text{refl}\}$ if $a_1$ and $\rho a_2$ parallel-reduce to a common expression, and as the empty set otherwise. We inductively define the judgment $\Gamma \vdash \rho : \{\text{refl}\}$, which asserts that $\rho$ maps to values in the correct interpretation, by

\[
\begin{array}{l}
\Gamma \vdash \rho : \emptyset \quad \Gamma, x \vdash \alpha : A \\
\Gamma \vdash \rho : \{\text{refl}\} \quad \Gamma \vdash \rho : \{\text{refl}\} \\
\Gamma, x \vdash \alpha : A \\
\Gamma \vdash \rho : \{\text{refl}\} \quad \Gamma \vdash \rho : \{\text{refl}\} \\
\Gamma, x \vdash \alpha : A
\end{array}
\]

Intuitively, $\Gamma \vdash \rho : \{\text{refl}\}$ asserts that $\rho$ maps term variables to well-behaved values. Because of the premise $\Gamma \vdash \alpha : \star$, it also asserts that $\Gamma$ does not contain any type variables. This is vacuously true for the empty context, and preserved by each case of the type interpretation.

In a normalization proof for System F or for CC \cite{Hilbe05}, the type interpretation would take an input $\rho$ which specifies the interpretation of type variables in $A$, but not one which specifies the values of term variables. Since we do not have polymorphism in our language, we do not need to account for type variables. But unlike CC, because of the primitive equality type we can not just ignore term variables in types. Our $\rho$ is similar to normalization proofs for systems that have large elimination of datatypes, such as CIC \cite{Nipkow:02}.

The soundness theorem relies on a few key lemmas about the interpretation. The first is a standard “downward closure” property for step-indexed relations: it says that requiring values to stay well-behaved for a larger number of steps creates a more precise interpretation.

**Lemma 4.** For any $A$, $\theta$ and $\rho$, if $j < k$ then $V^\rho_A \subseteq V^\rho_A$.

The next two lemmas are specific to $\lambda^\mathcal{T}$ because they relate the $\mathcal{L}$ and $\mathcal{P}$ interpretations of a type. They are used to handle the $\mathsf{TSUB}$ and $\mathsf{TMVAL}$ rules, respectively. The first says that the set of logical values is a subset of the corresponding programmatic sets.

**Lemma 5.** For any $A$, $k$, $\theta$ and $\rho$, $V^\rho_A \subseteq V^\rho_A$ and $C^\rho_A \subseteq C^\rho_A$.

The second says that for mobile types, the reverse containment also holds. For these types, the interpretations contain the same values in both fragments.

**Lemma 6.** For any $k$ and $\rho$, if $\text{Mobile}(A)$ then $V^\rho_A \subseteq V^\rho_A$.

Finally, for the $\mathsf{TCONV}$ rule, we need equal types to have the same interpretation.

**Lemma 7.** Suppose $\rho B_1 \Rightarrow^* A$ and $\rho B_2 \Rightarrow^* A$ and $\Gamma \vdash \rho : B_1$ and $\Gamma \vdash \rho : B_2$ and $\Gamma \vdash \rho : \{\text{refl}\}$. Then $a \in I_{\rho} B_1$ and $a \in I_{\rho} B_2$.

We can now prove soundness by induction on $\Gamma \vdash \rho : \alpha : A$. Normalization is an immediate corollary. We also get a characterization of which terms can be proven equal in the empty context. We need such a characterization to prove progress.

**Theorem 8 (Soundness).** If $\Gamma \vdash \rho : \alpha : A$ and $\Gamma \vdash \rho : \{\text{refl}\}$ and $\Gamma \vdash \rho : \{\text{refl}\}$.

**Corollary 9 (Normalization).** If $\Gamma \vdash \rho : \alpha : A$, then there exists a value $\nu$ such that $\alpha \equiv^* \nu$.

**Corollary 10 (Soundness of propositional equality).**

If $\Gamma \vdash \rho : \alpha : A_1 = A_2$, then there exists some $A$ such that $A_1 \Rightarrow^* A$ and $A_2 \Rightarrow^* A$. 
Normalization holds only for closed terms. This is a result of the fact that uses of the TCONV rule are unmarked in the syntax. It is possible to assume a contradictory equality and use it to typecheck a non-terminating term in the logical fragment. For example, the following statement is derivable:

$$y : \text{Nat} = (\text{Nat} \to \text{Nat})\downarrow (\lambda x.x \to x) \, (\lambda x.x) : \text{Nat}$$

This distinguishes \(\text{Nat}\) from intensional type theories like Coq and Agda. In those systems, our rule TCONV arises as the pattern-matching elimination form for a defined equality datatype. Uses of this eliminator would appear in the term above, and their reduction (elided). In the TCONV language with general recursion some explicit proofs are unavoidable, since they may diverge, so uses of \text{refl}

```plaintext
\text{Core language (ZT)}
```

Theorem 11 (Progress). If \(\vdash v : A\), then either \(v\) is a value, or there exists \(a'\) such that \(a \leadsto a'\).

5. Implementation

We have implemented a prototype dependently-typed language, called \text{Zombie}, based on \(\text{Nat}\). We have used this implementation to gain experience with the features described in this paper. Indeed, all of the example code in this paper can be type-checked by our implementation. These, and other examples are available for download.

Our language includes several features which were left out of \(\text{Nat}\) to keep the normalization proof simple. Instead of a single sort *, \text{Zombie} includes a full predicative hierarchy, which allows both polymorphism and type-level functions. We also include a general form for parameterized recursive datatypes, which subsumes \text{Nat}, \(A + B\), \(\Sigma x : A \cdot B\) and \(\mu x.A\). Datatypes are always mobile, and \text{Zombie} provides structural induction for all strictly
positive datatypes (not just Nat) following [20]. Finally, Zombie distinguishes between computationally relevant and irrelevant arguments [21], and includes a multiplace conversion operator, called *multiconversion* [22].

Adding these features to $\lambda^p$ would complicate the type interpretation, increasing the complexity of our machine-checked proof far beyond its current state. In particular, to add predicative polymorphism and type-level computation we would have to redefine our type interpretation as an induction over typing derivations, which is very painful to do in Coq. However, based on work in progress, we are optimistic that the metatheoretic requirements of these additional features will have little interaction with the fundamental consistency mechanism proposed here.

The general structure of our implementation appears in Figure 6. The part of our implementation that most closely resembles $\lambda^p$ is the internal language ZT. This language defines the operational behavior of Zombie expressions. However, like $\lambda^p$, type checking is not decidable for ZT expressions. Therefore, the implementation also includes an *annotated* version of ZT that supports syntax-directed type checking, an approach we have explored in [34].

Zombie is a direct representation of ZT typing derivations, marking all uses of conversion, subsumption, cumulativity, and coercion to and from A@B types. Furthermore, because reduction may not terminate, annotations on refl control and limit the search for a common reduct when proving that two terms are equal.

Directly working with ZT derivations incurs a considerable annotation burden for programmers. Therefore, the Zombie surface language makes these annotations optional. We are currently experimenting with a number of elaboration strategies to infer these annotations. These include using bidirectional type checking [31] to propagate type information through terms, unification to automatically infer some dependent arguments, and *congruence closure* [22] to automatically infer equality proofs used in conversions.

For example, consider the projection functions (fst and snd) for dependent pairs shown below. These functions pattern match their argument and return its first and second components respectively.

```
data Sigma (A:Type) (B:A -> Type) : Type where
    Pair of (x:A) (y : B x)
```

```
log fst : [A:Type] => [B:A -> Type] => Sigma A B -> A
fst (A) (B) p = case p of
    Pair x y -> x
```

```
log snd : [A:Type] => [B:A -> Type] => (p:Sigma A B) -> B (fst p)
snd (A) (B) p = case p of
    Pair x y -> unfold (fst p) in y
```

In the implementation of snd, unification can infer the arguments A and B to fst (which were marked inferable by the fat arrow ⇒ in the declaration of fst). Because not all expressions terminate, the programmer must explicitly ask the type checker to unfold (fst p) by β-reduction, which introduces the equation (fst p) = x into the context. That equation is then automatically used to convert the type of y from B x to B (fst p).

The examples we have implemented fall into two categories. The first includes the division and SAT-solving programs described in Section 1. These examples illustrate how one can write proofs about general recursive programs, and how general recursive programs can return proofs. Second, we have implemented functions for length-indexed lists (Vectors), finite sets represented as binary search trees, and data compression using run-length encoding, together with proofs of their correctness. Since these functions use simple structural recursion, they can be done entirely in the logical fragment. They show that although our core language requires annotations on refl and conv, the overhead of these annotations is manageable.

6. Related Work

In previous work, we introduced the proof technique of hybrid step-indexed/traditional logical relations, but for a simply-typed language [12]. This paper extends the normalization proof to a more expressive type system with dependent function types, an equality type, and conversion. It also improves the treatment of $\omega$-types by making them implicit. This change complicates the metatheory (see Lemma 8) but makes the language more expressive and simplifies the application rule.

**Terminating Sublanguage.** There are other dependently-typed languages which allow general recursion but identify a sublanguage of terminating expressions. Aura [18] and F* [38] do this using the kind system: expressions whose type has kind Prop are checked for normalization. Types can contain values but not non-value expressions, so there is no way to write separate proofs about programs. There also is no facility to treat programmatic values as proofs, e.g. a logical case expression cannot destruct a value from the nonterminating fragment.

ATS [13], GURU [36], and Sep3 [20] are dependently-typed languages where the logical and programmatic fragments are syntactically separate—in effect rejecting the rule TSub. One of the gains of this separation is that the logical language can be made CBN even though the programmatic one is CBV, avoiding the need for thunking (as discussed in Section 3). To do inductive reasoning, the Sep3 language adds an explicit “terminates” predicate.

Idris [10] is a full-spectrum dependently typed programming language that permits non-total definitions. Internally, it applies a syntactic test to check if function definitions are structurally decreasing, and programmers may ask whether particular definitions have been judged total. The type checker will only reduce expressions that have been proved terminating, again precluding separate equational reasoning about partial programs. Idris’ metatheory has not been studied formally.

**Partiality Monad.** Capretta’s partiality monad [11] uses coinductive types to embed general recursion into Type Theory. This approach treats pure functions as the default and nontermination less conveniently. Nonterminating programs must be written using monadic combinators (and are therefore never syntactically equal to pure programs). The partiality monad provides recursive function definitions but not general recursive types.

Furthermore, the coinductive approach requires a separate notion of equivalence to reason about partial programs. In, e.g., Coq, one would compare pure expressions according to the standard operational semantics, but define a coarser equivalence relation for partial terms that ignores the number of steps they take to normalize. Equations like $(\text{rec } f x.b) v = [v/x][\text{rec } f x.b/f/b]b$ do not hold with the usual Coq equality because the step counts differ. Conveninetly programming with equivalence relations like this, which are not directly justified by the reduction behavior of expressions, is an active area of research involving topics such as setoids [30], quotient types, and the univalence axiom [42].

**Non-constructive fixpoint semantics.** The work of Bertot and Komendatsky [9] describes a way to embed general recursive functions into Coq that does not use coinduction. They define a datatype partial $A$ that is isomorphic to the usual Maybe $A$ but is understood as representing a lifted CPO $A_\bot$, and use classical logic axioms to provide a fixpoint combinator fixp. When defining a recursive function the user must prove continuity side-conditions.
Since recursive functions are defined nonconstructively they cannot be reduced directly, so instead one must reason about them using the fix-point equation.

**Partial Types.** Nuprl has at its core an untyped lambda calculus, capable of defining a general fixed point combinator for defining recursive computations. In the core type theory, all expressions must be proven terminating when used. Constable and Smith [14] integrated potentially nonterminating computations through the addition of a type \( \overline{A} \) of partial terms of type \( A \). The fixpoint operator then has type \( \overline{A} \rightarrow \overline{A} \rightarrow \overline{A} \). However, to preserve the consistency of the logic, the type \( A \) must be restricted to admissible types. Crary [15] provides an expressive axiomatization of admissible types, but these conditions lead to significant proof obligations, especially when using \( \Sigma \)-types.

Smith [33] provides an example which shows that Nuprl needs this restriction. Writing \( a \downarrow \) for “\( a \) terminates”, define a \( \Sigma \)-type \( T \) of functions which are not total, and recursively define a \( p \) which inhabits \( T \):

\[
\begin{align*}
\text{Total } (f : \mathbb{N} \rightarrow \overline{\mathbb{N}}) & \equiv (n : \mathbb{N}) \rightarrow (f n) \downarrow \\
T & \equiv \Sigma(f : \mathbb{N} \rightarrow \overline{\mathbb{N}}). \text{Total } f \rightarrow \text{False} \\
(p : T) & \equiv \text{fix} (\lambda p. (g, \lambda h. \longrightarrow)) \\
g & \equiv \lambda x. \text{if } x = 0 \text{ then } 0 \text{ else } \pi_1(p)(x - 1)
\end{align*}
\]

Here the dash is an (elided) proof which sneakily derives a contradiction using \( \pi_2(p) \) and the hypothesis \( h \) that \( g \) is total. On the other hand, a separate induction shows that \( \pi_1(p) \) is total; it returns 0 for all arguments. This is a contradiction.

\( \lambda^\theta \) has almost all the ingredients for this paradox. Instead of a recursively defined pair we can use a recursive function Unit \( \rightarrow T \), and we can encode \( a \downarrow \) as \( \Sigma(y : A). a = y \). What saves us is that the proof in the second component of \( p \) uses the following reasoning principle: if \( \pi_1(p) \) terminates, then \( p \) terminates. In Nuprl \( a \downarrow \) is a primitive predicate and this inversion principle is built in. But using our encoding, a function \( (\pi_1(p) \downarrow) \rightarrow (p \downarrow) \) would have to magically guess the second component of a pair knowing only the first component. If we assume this function as an axiom we can encode the paradox and derive inconsistency, so our consistency proof shows that there is no way to write such a function.

**Hoare Type Theory.** HTT [26, 37] is another embedding of general programs into a type theory like Coq, which goes beyond non-termination to also handle memory effects. Instead of a unary type constructor \( A \), it adds the indexed type \( \{P \mid x : A : \{Q \} \} \) representing an effectful computation returning \( A \) and with pre- and postconditions \( P \) and \( Q \). The assertions \( P \) and \( Q \) can use all of Coq, so the type of a computation can specify its behavior precisely. However, computations can not be evaluated during type checking (the fixpoint combinator and memory access primitives are implemented as Coq axioms with types but no reduction rules).

**Fixpoint induction** Domain-theory based formalisms provide two basic reasoning principles for proving properties about recursive functions: unfolding a function definition, and fixpoint induction. The latter principle (see e.g. [44]) states that to prove a property about a function, one may assume it as an induction hypothesis for the recursive calls of the function. For this to be valid, the property must be “admissible”, and it must hold for infinite loops. An equivalent variant [9] is to allow induction on the number of recursive steps an expression takes to normalize.

\( \lambda^\theta \) currently provides no such principle. If a theorem can not be proved just from unfolding, there are two ways to proceed. In order to prove \( \text{div}_1 \text{le} \) in Section 1 we used (strong) natural-number induction. For this strategy to work the programmer has to find a termination metric for the function in question, so it only works for functions that are in fact terminating. However, it can still be convenient to give a direct recursive definition of the function. For functions that genuinely do not terminate, one can instead change them to return a \( \Sigma \)-type asserting the property, so that the property is automatically available for recursive calls. This is what we did for \( \text{selver} \) in Section 1 and it is the only option in Hoare Type Theory.

**Modal types for distributed computation.** Modal logic reasons about statements whose truth varies in different “possible worlds”. Our type system is formally similar, with the possible worlds being L and P. Modal logic has previously been used to design type systems for distributed computation [19, 25]. In particular, \( \lambda^\theta \) was inspired by ML5 [25], in which the typing judgment is indexed by what “world” (computer in a distributed system) a program is running on, and which includes a type \( A@\theta \) internalizing that judgment. Our rule TMV is similar to the get rule in ML5, and our Mobile (A) is similar to the judgment A mobile in ML5. On the other hand, unlike \( \lambda^\theta \), ML5 does not require that the domain of an arrow type be mobile. As we explained in Section 3.1 we make that restriction to accommodate our rule TSUB, a rule which does not make sense in the context of distributed computation.

**7. Future work**

In future work, we hope to extend the metatheory of \( \lambda^\theta \) to include more of ZT. We plan to allow polymorphic types and type-level functions in both the L and P fragments, extending our proof using ideas from normalization proofs for the Calculus of Constructions [16]. Following the ideas of Ahn and Sheard [2] and their language Nax [3], we also hope to add combinators to define recursive functions over recursive data to the logical language. Nax places no restriction on what sorts of datatypes can be defined or how they can be constructed. Instead, it limits the analysis of data structures to ensure the soundness of the logic. More generally, we would like to extend our proofs to a general theory of datatype definitions, maybe encoded via recursion, sums, and products as in ID2 [4]. One potential worry is that we assume injectivity for all type constructors, which can be used to encode Cantor-like paradoxes. We hope to avoid inconsistency by forbidding impredicative polymorphism and datatypes with “large” indices.

Adding these features will require substantial additional work in the normalization proof, but we do not anticipate any changes to the novel typing rules that connect the L and P fragments.

**Reasoning about general recursive functions** Currently \( \lambda^\theta \) emphasizes lightweight verification. In order to turn it into a tool for full verification of potentially nonterminating programs, we would add stronger reasoning principles.

First, the value restrictions in \( \sim \) can get in the way of equational reasoning. If \( a \) is an expression in \( P \) there is no way to prove an equation like \( (\text{let } x = a \text{ in } f x) = (f a) \), even though the two sides are in fact contextually equivalent. To make it provable we could add termination-case—a case analysis on whether a programmatic expression evaluates to a value or diverges [29]. Unfortunately, this operator is unimplementable, so we would not want to allow proofs that use this reasoning to be used as programs. One solution is to introduce a new consistency classifier \( \Omega \), for oracular, in addition to L and P. By not allowing \( \Omega \) expressions to be used as programs, we could control and track the use of termination case.

Second, we would like to investigate whether some (perhaps weakened) form of fixpoint induction can be consistently added. The experience with partial types in Nuprl suggests that this may require a notion of admissible predicates.
8. Conclusion

This paper presents a framework for interacting logics and programming languages. The consistency classifiers, , describe the set of typing rules that determine the properties of each well-typed expression. At the same time, many standard typing rules are polymorphic in this classifier, leading to uniformity between the systems. Internalizing this judgment as a type and observing that some values can move freely allows the fragments to interact in nontrivial ways, leading to an expressive foundation for dependently-typed programming.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. 0910500 and 1116620. The Zombie implementation was developed with the assistance of the Trellys team, and the ideas in this paper benefitted greatly from that collaboration. This paper was written with the help of the excellent team, and the ideas in this paper benefitted greatly from their considered and helpful comments.

References


