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 [33] 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 heterogenous 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 $A@\theta$. 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 λ^{θ}, that supports these interactions (see Sections 2 and 3). In addition to the $A@\theta$ type, this language includes dependent functions, products, propositional equality, natural numbers, sums, recursive functions

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535883

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[1].

We also explore how our ideas interact with other programming language features. We have implemented a prototype language, `Zombie`, based on the semantics of λ^θ, and discuss that implementation in Section 5. `Zombie` extends λ^θ 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[2].

We are not the first to consider the combination of total and partial programming in the setting of dependently-typed languages. Partial types [14] and the coinductive partiality monad [11] embed general recursive programs into constructive logic by modeling nontermination. Alternatively, languages such as Idris [10], Aura [18], and F* [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 λ^θ, 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 **`log`** in its definition:

```
log not : Bool → Bool
not b = if b then False else 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 `n_eq` is a proof that `n'` is a subterm of `n`.

```
log plus : Nat → Nat → Nat
ind plus n m =
  case n [n_eq] of
    Zero    → m
    Succ n' → Succ (plus n' n_eq m)
```

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

```
prog div : Nat → Nat → Nat
rec div n m = if lt n m then 0
              else plus 1 (div (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 `lt` and `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.

```
data Maybe (A : Type) where
  Nothing
  Just 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.

```
log md3 : Maybe (Nat → Nat)
md3 = Just (\x. div 3 x)

log foo : Maybe (Nat → Nat) → (Nat → Nat @ P)
foo x = case x of
  Just y  → y
  Nothing → \x. x
```

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

```
prog bar : Maybe (Nat → Nat) → Nat → Maybe Nat
bar x y = case x of
      Just f  → Just (f y)
      Nothing → Nothing

prog boom : Maybe Nat
boom = bar md3 0
```

Proofs about programs. Having defined the programmatic function `div`, we might wish to verify facts about it. As a simple example, we prove that `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.

```
log div63 : div 6 3 = 2
div63 = refl
```

The proof above (**`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. **`refl`** `5000`). In languages like Aura or F*, this theorem cannot even be stated because non-value expressions such as `div 6 3` cannot appear in types. This example illustrates an important property of our language, which we call *freedom of speech*: although proofs cannot themselves use general recursion, they are allowed to *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:

```
log div_le : (n:Nat) → (m:Nat) → (eq m 0 = False)
         → (le (div n m) n = True)
```

Above, `eq` is an equality function for natural numbers and `le m n` determines whether $m \leq n$. We do not show proof of the above theorem here, though it is available with our implementation. The

[1] Proofs available at `http://www.cis.upenn.edu/~ccasin/papers/combining-coq.tgz`

[2] Implementation available at `https://code.google.com/p/trellys` in the branch `branches/zombie-trellys-POPL14/`

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.

```
data Ans (n : Nat) (form : Formula n) : Type where
  SAT    of (assign : Vector (Maybe Bool) n)
             (proof  : satisfies assign form =
                       (Just True : Maybe Bool))
  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. `Zombie` can automatically fill in the _ below with the proof that `assign` 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.

```
prog solver : (n:Nat) → (formula : Formula n)
            → Vector (Maybe Bool) n
            → Ans n formula @ L
solver n formula assign =
  case (satisfies assign formula) of
    Just True  → SAT assign _
    Just False → UNSAT
    Nothing    → ....
```

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 *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.

```
let prog f     = (... : Formula n) in
let log  empty = repeat (Nothing : Maybe Bool) n in
let prog isSat = (solver n f empty : Ans n f @L) in
let log  prf   = case isSat of
    SAT assignment pf →
        -- ... use proof of satisfiability ...
    UNSAT  → ...
```

$$
\begin{array}{lcl}
\theta & ::= & \mathsf{L} \mid \mathsf{P} \\
a,\ b,\ A,\ B & ::= & \star \mid (x : A) \to B \mid a = b \\
 & \mid & \mathsf{Nat} \mid A + B \mid \Sigma x{:}A.B \mid \mu x.A \mid A@\theta \\
 & \mid & x \mid \lambda x.a \mid \mathsf{rec}\ f\ x.a \mid \mathsf{ind}\ f\ x.a \mid a\ b \\
 & \mid & \mathsf{refl} \mid \mathsf{inl}\ a \mid \mathsf{inr}\ b \\
 & \mid & \mathsf{scase}_z\ a\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1 ; \mathsf{inr}\ y \Rightarrow a_2\} \\
 & \mid & \langle a, b\rangle \mid \mathsf{pcase}_z\ a\ \mathsf{of}\ \{(x,\ y) \Rightarrow b\} \\
 & \mid & \mathsf{Z} \mid \mathsf{S}\ a \mid \mathsf{ncase}_z\ a\ \mathsf{of}\ \{Z \Rightarrow a_1 ; \mathsf{S}\ x \Rightarrow a_2\} \\
 & \mid & \mathsf{roll}\ a \mid \mathsf{unroll}\ a \\
 & & \\
v & ::= & \star \mid (x : A) \to B \mid a = b \\
 & \mid & \mathsf{Nat} \mid A + B \mid \Sigma x{:}A.B \mid \mu x.A \mid A@\theta \\
 & \mid & x \mid \lambda x.a \mid \mathsf{rec}\ f\ x.a \mid \mathsf{ind}\ f\ x.a \mid \mathsf{refl} \\
 & \mid & \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \langle v_1, v_2\rangle \mid \mathsf{Z} \mid \mathsf{S}\ v \mid \mathsf{roll}\ v
\end{array}
$$

$\boxed{a \rightsquigarrow b}$

$$\frac{}{(\lambda x.a)\ v \rightsquigarrow [v/x]a}\ \textsc{SLam}$$

$$\frac{}{(\mathsf{rec}\ f\ x.a)\ v \rightsquigarrow [v/x][\mathsf{rec}\ f\ x.a/f]a}\ \textsc{SFun}$$

$$\frac{}{(\mathsf{ind}\ f\ x.a)\ v \rightsquigarrow [v/x][\lambda y.\lambda z.(\mathsf{ind}\ f\ x.a)\ y/f]a}\ \textsc{SInd}$$

$$\frac{}{\mathsf{scase}_z\ \mathsf{inl}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1 ; \mathsf{inr}\ x \Rightarrow a_2\} \rightsquigarrow [\mathsf{refl}/z][v/x]a_1}\ \textsc{SCL}$$

$$\frac{}{\mathsf{scase}_z\ \mathsf{inr}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1 ; \mathsf{inr}\ x \Rightarrow a_2\} \rightsquigarrow [\mathsf{refl}/z][v/x]a_2}\ \textsc{SCR}$$

$$\frac{}{\mathsf{pcase}_z\ \langle v_1, v_2\rangle\ \mathsf{of}\ \{(x,\ y) \Rightarrow a\} \rightsquigarrow [\mathsf{refl}/z][v_1/x][v_2/y]a}\ \textsc{SCP}$$

$$\frac{}{\mathsf{unroll}\ (\mathsf{roll}\ v) \rightsquigarrow v}\ \textsc{SUnroll}$$

Figure 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 `Nat` can never cause divergence, so it is safe to be used in logical expressions even when not marked as `@L`. Similarly, the `Ans` type above is also mobile, so the `@L` annotation on the type of `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 λ^θ language

We begin our technical development with an overview of the formal language, λ^θ. 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 $\star$ (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 λ^θ 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 (SInd) is described in Section 2.1. To save space, most rules have been omitted. The full set of rules can be found in the companion technical report[3].

Values include the standard components of functional programming: recursive functions $\mathsf{rec}\ f\ x.a$, nonrecursive functions $\lambda x.a$, natural numbers (constructed by Z and $\mathsf{S}\ a$ and eliminated by ncase), disjoint unions (constructed by $\mathsf{inl}\ a$ and $\mathsf{inr}\ a$ and eliminated by scase), dependently typed pairs (constructed by $\langle a, b\rangle$ and eliminated by pcase), and recursive data (introduced by $\mathsf{roll}\ a$ and eliminated by $\mathsf{unroll}\ a$). Values also include $\star$, all type forms, a trivial equality proof refl, and variables. Including variables is safe because CBV evaluation only substitutes values for variables and it is useful because it allows the λ^θ type checker to reduce open terms.

We chose CBV because of its simple cost model, but this choice also affects the interaction between the logical and programmatic fragments. As shown in Sections 2.2 and 3.3, the type system takes advantage of the fact that values cannot induce nontermination. As a result, some typing rules apply only to values.

Note that expressions do not contain type annotations. Types describe terms but do not interfere with equality. 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 λ^θ term. This is not a problem because we do not intend programmers to write these terms directly. Instead, our implementation uses an annotated *surface* language that the type checker elaborates into typing derivations (see Section 5).

The rest of this section describes the specific details of λ^θ, 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 λ^θ is a dependent type theory where the typing judgment $\Gamma \vdash^\theta a : A$ is indexed by a consistency classifier θ. The judgement is designed so that expressions that type check at L 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 $\Gamma \vdash^\theta A : \star$.

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 θ 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-kindedness. For example, TARR checks a function type by checking the the domain and range. We discuss the premise $\mathsf{Mobile}\,(A)$, which asserts that A is a mobile type, in Section 3.3.

There are three ways to define functions in λ^θ. Rule TLAM types non-recursive λ-expressions in the logical fragment, whereas rule TREC types general recursive rec-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 $(\mathsf{ind}\ f\ x.b)$, the recursive

[3] Available from `http://www.seas.upenn.edu/~ccasin/papers/combining-TR.pdf` and as University of Pennsylvania CIS Technical Report MS-CIS-13-08.

$\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \cdot}\textsc{CNil} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, x :^\theta \star}\textsc{CStar} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash^\theta A : \star}{\vdash \Gamma, x :^\theta A}\textsc{CType}$$

$\boxed{\Gamma \vdash^\theta a : A}$

$$\frac{(x :^\theta A) \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash^\theta x : A}\textsc{TVar} \qquad \frac{\begin{array}{c}\Gamma \vdash^\theta A : \star \quad \mathsf{Mobile}\,(A)\\ \Gamma, x :^\theta A \vdash^\theta B : \star\end{array}}{\Gamma \vdash^\theta (x : A) \to B : \star}\textsc{TArr}$$

$$\frac{\begin{array}{c}\Gamma \vdash^\theta b : (x : A) \to B\\ \Gamma \vdash^\theta a : A \quad \Gamma \vdash^\theta [a/x]B : \star\end{array}}{\Gamma \vdash^\theta b\ a : [a/x]B}\textsc{TApp}$$

$$\frac{\begin{array}{c}\Gamma, x :^\mathsf{L} A \vdash^\mathsf{L} b : B\\ \Gamma \vdash^\mathsf{L} (x : A) \to B : \star\end{array}}{\Gamma \vdash^\mathsf{L} \lambda x.b : (x : A) \to B}\textsc{TLam}$$

$$\frac{\begin{array}{c}\Gamma, f :^\mathsf{P} (x : A) \to B, x :^\mathsf{P} A \vdash^\mathsf{P} b : B\\ \Gamma \vdash^\mathsf{P} (x : A) \to B : \star\end{array}}{\Gamma \vdash^\mathsf{P} \mathsf{rec}\ f\ x.b : (x : A) \to B}\textsc{TRec}$$

$$\frac{\begin{array}{c}\Gamma, x :^\mathsf{L} \mathsf{Nat}, f :^\mathsf{L} (y : \mathsf{Nat}) \to (z : \mathsf{S}\ y = x) \to B \vdash^\mathsf{L} b : B\\ \Gamma \vdash^\mathsf{L} (x : \mathsf{Nat}) \to B : \star\end{array}}{\Gamma \vdash^\mathsf{L} \mathsf{ind}\ f\ x.b : (x : \mathsf{Nat}) \to B}\textsc{TInd}$$

$$\frac{\begin{array}{c}\Gamma \vdash^\theta A : \star\\ \Gamma \vdash^\theta B : \star\end{array}}{\Gamma \vdash^\theta A + B : \star}\textsc{TSum}$$

$$\frac{\begin{array}{c}\Gamma \vdash^\theta a : A\\ \Gamma \vdash^\theta A + B : \star\end{array}}{\Gamma \vdash^\theta \mathsf{inl}\ a : A + B}\textsc{TInl} \qquad \frac{\begin{array}{c}\Gamma \vdash^\theta b : B\\ \Gamma \vdash^\theta A + B : \star\end{array}}{\Gamma \vdash^\theta \mathsf{inr}\ b : A + B}\textsc{TInr}$$

$$\frac{\begin{array}{c}\Gamma \vdash^\theta a : A_1 + A_2 \quad \Gamma \vdash^\theta B : \star\\ \Gamma, x :^\theta A_1, z :^\mathsf{L} \mathsf{inl}\ x = a \vdash^\theta b_1 : B\\ \Gamma, x :^\theta A_2, z :^\mathsf{L} \mathsf{inr}\ x = a \vdash^\theta b_2 : B\end{array}}{\Gamma \vdash^\theta \mathsf{scase}_z\ a\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow b_1; \mathsf{inr}\ x \Rightarrow b_2\} : B}\textsc{TSCase}$$

$$\frac{\begin{array}{l}\Gamma \vdash^\theta A : \star\\ \mathsf{Mobile}\,(A)\\ \Gamma, x :^\theta A \vdash^\theta B : \star\end{array}}{\Gamma \vdash^\theta \Sigma x{:}A.B : \star}\textsc{TSigma} \qquad \frac{\begin{array}{l}\Gamma \vdash^\theta \Sigma x{:}A.B : \star\\ \Gamma \vdash^\theta a : A\\ \Gamma \vdash^\theta b : [a/x]B\\ \Gamma \vdash^\theta [a/x]B : \star\end{array}}{\Gamma \vdash^\theta \langle a, b\rangle : \Sigma x{:}A.B}\textsc{TPair}$$

$$\frac{\begin{array}{c}\Gamma \vdash^\theta a : \Sigma x{:}A_1.A_2 \quad \Gamma \vdash^\theta B : \star\\ \Gamma, x :^\theta A_1, y :^\theta A_2, z :^\mathsf{L} \langle x, y\rangle = a \vdash^\theta b : B\end{array}}{\Gamma \vdash^\theta \mathsf{pcase}_z\ a\ \mathsf{of}\ \{(x,\ y) \Rightarrow b\} : B}\textsc{TPCase}$$

$$\frac{\Gamma, x :^\mathsf{L} \star \vdash^\mathsf{L} A : \star}{\Gamma \vdash^\mathsf{L} \mu x.A : \star}\textsc{TMu} \qquad \frac{\begin{array}{c}\Gamma \vdash^\theta a : [\mu x.A/x]A\\ \Gamma \vdash^\theta \mu x.A : \star\end{array}}{\Gamma \vdash^\theta \mathsf{roll}\ a : \mu x.A}\textsc{TRoll}$$

$$\frac{\Gamma \vdash^\mathsf{P} a : \mu x.A \quad \Gamma \vdash^\mathsf{P} [\mu x.A/x]A : \star}{\Gamma \vdash^\mathsf{P} \mathsf{unroll}\ a : [\mu x.A/x]A}\textsc{TUnroll}$$

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 SIND from Figure 1).

The rule for function application, TAPP, differs from the usual application rule in pure dependently-typed languages in the additional third premise $\Gamma \vdash^{\theta} [a/x]B : s$, which checks that the result type is well-formed. Some rules of the language (such as β-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 plus_zero : (n:Nat) → plus n 0 = n
```

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

```
plus_zero (f x) : plus (f x) 0 = (f x)
```

The rules for sum types (TSUM, TINL, TINR, and TSCASE) provide dependent case analysis. The term scase 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 (TSIGMA, TPAIR, 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 $\Gamma \vdash^{\theta} [a/x]B : \star$ ensures that substituting the expression a does not violate any assumptions made about the value x 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 Σ-types are derivable.

Finally, the rules TMU, TROLL and 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 $\mu x.(x \to \mathsf{Nat})$—are a potential source of nontermination. To ensure normalization, it suffices to restrict the the elimination rule TUNROLL to be in P. The introduction rule 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.

$$\boxed{\Gamma \vdash^{\theta} a : A} \qquad \frac{\Gamma \vdash^{\mathsf{P}} a : A \qquad \Gamma \vdash^{\mathsf{P}} b : B}{\Gamma \vdash^{\mathsf{L}} a = b : \star}\ \text{TEQ}$$

$$\frac{\begin{array}{c} a \Rrightarrow^* c \qquad b \Rrightarrow^* c \\ \Gamma \vdash^{\theta_1} a : A \qquad \Gamma \vdash^{\theta_2} b : B \end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : a = b}\ \text{TREFL}$$

$$\frac{\begin{array}{c} \Gamma \vdash^{\mathsf{L}} b : b_1 = b_2 \qquad \Gamma \vdash^{\theta} a : [b_1/x]A \\ \Gamma \vdash^{\theta} [b_2/x]A : \star \end{array}}{\Gamma \vdash^{\theta} a : [b_2/x]A}\ \text{TCONV}$$

Figure 3. Typing: equality

Therefore, the rule 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 refl is the primitive proof of equality. Rule TREFL says that 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 $a \Rrightarrow b$. This relation extends the ordinary evaluation $a \rightsquigarrow b$ by allowing reduction under binders, e.g. $(\lambda x.1 + 1) \Rrightarrow (\lambda x.2)$ 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 TCONV. The proof term is checked in 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, $\mathsf{Nat} = \mathsf{Nat} \to \mathsf{Nat}$, it would be easy to create a stuck term. Similar to 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 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 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 λ^{θ} 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 programmatically. We reflect this fact into the type system by the rule TSUB, which says that if a term a type checks logically, then it will also type check programmatically. 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

$\boxed{\Gamma \vdash^{\theta} a : A}$

$$\frac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{P}} a : A}\textsc{TSub} \qquad \frac{\Gamma \vdash^{\theta'} A : \star}{\Gamma \vdash^{\theta} A@\theta' : \star}\textsc{TAt}$$

$$\frac{\Gamma \vdash^{\theta} v : A@\theta' \quad \Gamma \vdash^{\theta'} A : \star}{\Gamma \vdash^{\theta'} v : A}\textsc{TUnboxVal} \qquad \frac{\Gamma \vdash^{\theta} a : A \quad \Gamma \vdash^{\theta} A : \star}{\Gamma \vdash^{\mathsf{P}} a : A@\theta}\textsc{TBoxP}$$

$$\frac{\Gamma \vdash^{\mathsf{L}} a : A \quad \Gamma \vdash^{\theta} A : \star}{\Gamma \vdash^{\mathsf{L}} a : A@\theta}\textsc{TBoxL} \qquad \frac{\Gamma \vdash^{\mathsf{P}} v : A \quad \Gamma \vdash^{\mathsf{P}} A : \star}{\Gamma \vdash^{\mathsf{L}} v : A@\mathsf{P}}\textsc{TBoxLV}$$

Figure 4. Typing: subsumption and internalized consistency classification

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 $(x : A@\mathsf{L}) \to B$), return them as postconditions (with functions of type $(x : A) \to (B@\mathsf{L})$), and store them in data structures (with pairs of type $\Sigma x : A.(B@\mathsf{L})$). At the same time, logical lemmas can use @ to manipulate values from the programmatic fragment.

The rules for the $A@\theta$ type appear in Figure 4. Intuitively, the judgment $\Gamma \vdash^{\theta_1} a : A@\theta_2$ holds if the fragment θ_1 may safely observe that $\Gamma \vdash^{\theta_2} a : A$. This intuition is captured by the three introduction rules. The programmatic fragment can internalize any typing judgement (TBoxP), but in the logical fragment (TBoxL and TBoxLV) we sometimes need a restriction to ensure termination. Therefore, rule TBoxLV only applies when the subject of the typing rule is a value. (The rule TBoxL can introduce $A@\theta$ for any θ since logical terms are also programmatic). Both introduction and elimination of @ is unmarked 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 @L, e.g., $A@\mathsf{L} \to B$, 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:

$$\frac{\Gamma \vdash^{\mathsf{P}} f : A@\mathsf{L} \to B \quad \dfrac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{P}} a : A@\mathsf{L}}\textsc{TBoxP}}{\Gamma \vdash^{\mathsf{P}} f\ a : B}\textsc{TApp}$$

$$\frac{\Gamma \vdash^{\mathsf{L}} g : A@\mathsf{P} \to B \quad \dfrac{\Gamma \vdash^{\mathsf{P}} v : A}{\Gamma \vdash^{\mathsf{L}} v : A@\mathsf{P}}\textsc{TBoxLV}}{\Gamma \vdash^{\mathsf{L}} g\ v : B}\textsc{TApp}$$

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: Nat) → (f: (Nat → Nat)@P) → (f (plus n 0) = f n)
```

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

$\boxed{\mathsf{Mobile}\,(A)}$

$$\frac{}{\mathsf{Mobile}\,(A@\theta)}\textsc{MAt}$$

$$\frac{}{\mathsf{Mobile}\,(a = b)}\textsc{MEq} \qquad \frac{\mathsf{Mobile}\,(A) \quad \mathsf{Mobile}\,(B)}{\mathsf{Mobile}\,(\Sigma x{:}A.B)}\textsc{MSigma}$$

$$\frac{}{\mathsf{Mobile}\,(\mathsf{Nat})}\textsc{MNat} \qquad \frac{\mathsf{Mobile}\,(A) \quad \mathsf{Mobile}\,(B)}{\mathsf{Mobile}\,(A + B)}\textsc{MSum}$$

$\boxed{\Gamma \vdash^{\theta} a : A}$

$$\frac{\Gamma \vdash^{\mathsf{P}} v : A \quad \Gamma \vdash^{\mathsf{L}} A : \star \quad \mathsf{Mobile}\,(A)}{\Gamma \vdash^{\mathsf{L}} v : A}\textsc{TMVal}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\theta} a : (A_1 + A_2)@\theta' \quad \Gamma \vdash^{\theta} B : \star \\ \Gamma, x :^{\theta'} A_1, z :^{\mathsf{L}} \mathsf{inl}\, x = a \vdash^{\theta} b_1 : B \\ \Gamma, x :^{\theta'} A_2, z :^{\mathsf{L}} \mathsf{inr}\, x = a \vdash^{\theta} b_2 : B\end{array}}{\Gamma \vdash^{\theta} \mathsf{scase}_z\ a\ \mathsf{of}\ \{\mathsf{inl}\, x \Rightarrow b_1; \mathsf{inr}\, x \Rightarrow b_2\} : B}\textsc{TSCase'}$$

$$\frac{\begin{array}{c}\Gamma \vdash^{\theta} a : (\Sigma x{:}A_1.A_2)@\theta' \quad \Gamma \vdash^{\theta} B : \star \\ \Gamma, x :^{\theta'} A_1, y :^{\theta'} A_2, z :^{\mathsf{L}} \langle x, y\rangle = a \vdash^{\theta} b : B\end{array}}{\Gamma \vdash^{\theta} \mathsf{pcase}_z\ a\ \mathsf{of}\ \{(x,\ y) \Rightarrow b\} : B}\textsc{TPCase'}$$

Figure 5. Typing: mobile types and cross-fragment case expressions

There is no way to apply a logical lemma to a programmatic *non*-value expression. If an expression `a` may diverge then so may `f a`, so we must not assign it a type in the logical fragment.[4] However, we can work around this restriction by either first evaluating `a` to a value in the programmatic fragment or by thunking.

The @-types are eliminated by the rule TUnboxVal. To preserve termination, the rule is restricted to apply only to values.

Recall the function `solver` of type

```
prog solver : (n:Nat) → (f:Formula n)
          → Vector (Maybe Bool) n → (Ans n f)@L
```

In the introduction, we asserted that the following code type checks.

```
let prog isSat = (solver n f empty : Ans n f @L) in
let log  prf = case isSat of
    SAT a pf → -- ... here pf is logical ...
    UNSAT    → -- ...
```

In this example, the logical program `prf` cannot directly treat `solver n f empty` as a proof because it may diverge. However, once it has been evaluated to a value, it can be safely used by the logical fragment. Above, the let binding forces evaluation of the expression `solver n f empty`, introducing a new programmatic variable `isSat : Ans n f @ L` into the context. Because variables are values, any logical context can freely use the variable through TUnboxVal even though it was computed by the programmatic language.

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 `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 $\mathsf{Mobile}\,(A)$.. Intuitively, a type is mobile if the same set of values inhabit the type when $\theta = \mathsf{L}$ and when $\theta = \mathsf{P}$. In particular, these types do not include functions (though any type may be made mobile by tagging its fragment with @).

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 θ 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 θ' to be eliminated in another fragment θ. This lets the logical language reason by case analysis on programmatic values. Similarly, TPCASE' 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 $((a = b)@\mathsf{L}) \to B$, since when needed, the body of the function can treat the argument as logical through TMVAL. Similarly, multiple @'s have no effect beyond the innermost @ in a type. Values of type $A@\mathsf{P}@\mathsf{L}@\mathsf{P}@\mathsf{L}@\mathsf{P}$ can be used as if they had type $A@\mathsf{P}$.

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

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

```
(f : (Nat → Nat)@P) → f (plus n 0) = f 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 @-type, as in `(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:

$$\dfrac{\dfrac{\Gamma, x :^{\mathsf{L}} A \vdash^{\mathsf{L}} b : B}{\Gamma \vdash^{\mathsf{L}} (\lambda x.b) : (x : A) \to B}\ \textsc{TLam}}{\Gamma \vdash^{\mathsf{P}} (\lambda x.b) : (x : A) \to B}\ \textsc{TSub}$$

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 $\mathsf{Mobile}\,(A)$ checks.

4. Metatheory

We now describe the metatheory of λ^θ. 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].

4.1 Preservation

As usual, the preservation proof relies on weakening, substitution and inversion lemmas. The weakening lemma is standard. Due to the value restrictions in the type system, the substitution lemma is restricted to values:

LEMMA 1 (Substitution). *If* $\Gamma_1, x :^{\theta'} B, \Gamma_2 \vdash^{\theta} a : A$ *and* $\Gamma_1 \vdash^{\theta'} v : B$, *then* $\Gamma_1, [v/x]\Gamma_2 \vdash^{\theta} [v/x]a : [v/x]A$.

However, our inversion lemmas are more complicated than usual, because one of the design goals of λ^θ is that typing rules without runtime effects should not require annotation. In particular, uses of TCONV and TBOXP/L/LV are not marked.

For example, consider inversion for λ-expressions. Usually, it is the case that if $\Gamma \vdash (\lambda x.b) : A$, then A is β-convertible with some arrow type $(x : B_1) \to B_2$ and $\Gamma, x : B_1 \vdash b : B_2$. In λ^θ this is not true: if there were a hypothesis $(x :^{\mathsf{L}} (\mathsf{Nat} \to \mathsf{Nat}) = \mathsf{Nat}) \in \Gamma$, the expression could also have been given type Nat using TCONV. (Restricting preservation to empty contexts would not help, since at this point in the proof—before proving consistency—we cannot rule out that this equality is provable). Alternatively, if the BOX rules were used, A may be an @-type. Taking this into account, our inversion lemma reads:

LEMMA 2 (Inversion for λ-expressions). *If* $\Gamma \vdash^{\theta} (\lambda x.b) : B$, *then there is some* p *and* $(x : B_1) \to B_2$ *such that either*

1. $\Gamma \vdash^{\mathsf{L}} p : B = ((x : B_1) \to B_2)$ *and* $\Gamma, x :^{\theta} B_1 \vdash^{\theta} b : B_2$,
2. *or there are some* $\theta' \ldots \theta''$ *such that* $\Gamma \vdash^{\mathsf{L}} p : B = (((x : B_1) \to B_2)@\theta' \ldots @\theta'')$ *and* $\Gamma, x :^{\theta'} B_1 \vdash^{\theta'} b : B_2$.

With this and other similar inversion lemmas, we can prove preservation.

THEOREM 3 (Preservation). *If* $\Gamma \vdash^{\theta} a : A$ *and* $a \rightsquigarrow a'$, *then* $\Gamma \vdash^{\theta} a' : A$.

The proof of the preservation theorem requires the addition of type constructor discrimination and injectivity rules (Figure 6) to the type system. The discrimination rule TCONTRA eliminates contradictory equalities. If we can prove a contradiction we must be in unreachable code, so we allow giving any typeable expression a any wellformed type B at any θ'.

An equation $B_1 = B_2$ counts as contradictory if the *head forms* of both sides are defined and unequal. The head form of a type is its outermost constructor. For example, the head form of $(x : A) \to B$ is $\to$, and the head form of Nat is Nat. The complete definition of $\mathsf{hd}(A)$ appears in the companion technical report.

The injectivity rules invert equality proofs between type forms. For example, from a proof $\Gamma \vdash^{\mathsf{L}} p : ((x : A_1) \to A_2) = ((x : B_1) \to B_2)$ we can also derive $\Gamma \vdash^{\mathsf{L}} p : A_1 = B_1$. Similar typing rules are available for @, sum and pair types. These are elided here for space, but included in the technical report.

These rules are necessitated by the weak inversion lemmas. Consider, e.g., the case when a function application beta reduces, $(\lambda x.b)\ v \rightsquigarrow [v/x]b$. From the premises of the rule TAPP we know that $\Gamma \vdash^{\theta} (\lambda x.b) : (x : A_1) \to A_2$ and $\Gamma \vdash^{\theta} v : A_1$, and from

$$\boxed{\Gamma \vdash^{\theta} a : A}$$

$$\frac{\Gamma \vdash^{\theta} a : ((x\!:\!A_1) \to A_2) = ((x\!:\!B_1) \to B_2) \qquad \Gamma \vdash^{\theta} A_1 = B_1 : \star}{\Gamma \vdash^{\theta} a : A_1 = B_1}\text{TARRINV1}$$

$$\frac{\Gamma \vdash^{\mathsf{L}} a_1 : B_1 = B_2 \qquad \mathsf{hd}(B_1) \neq \mathsf{hd}(B_2) \qquad \Gamma \vdash^{\theta} a : A \qquad \Gamma \vdash^{\theta} A : \star \quad \Gamma \vdash^{\theta'} B : \star}{\Gamma \vdash^{\theta'} a : B}\text{TCONTRA}$$

$$\frac{\Gamma \vdash^{\theta} a : ((x\!:\!A_1) \to A_2) = ((x\!:\!B_1) \to B_2) \qquad \Gamma \vdash^{\theta'} v : A_1 \quad \Gamma \vdash^{\theta} [v/x]A_2 = [v/x]B_2 : \star}{\Gamma \vdash^{\theta} a : [v/x]A_2 = [v/x]B_2}\text{TARRINV2}$$

Figure 6. Typing: discrimination and injectivity of type constructors (injectivity rules for @-, μ-, pair- and sum-types omitted).

inversion we know either $\Gamma \vdash^{\mathsf{L}} p : ((x : A_1) \to A_2) = ((x : B_1) \to B_2)$ and $\Gamma, x :^{\theta} B_1 \vdash^{\theta} b : B_1$, or else $(x : A_1) \to A_2$ is provably equal to an @-type. In the first case we apply the substitution lemma, using TARRINV1 to prove $A_1 = B_1$, while in the second case we use TCONTRA.

4.2 Normalization and Progress

Our proof of normalization builds upon the standard Girard-Tait reducibility method [17, 39] 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 $\mathcal{V}_\rho[\![A]\!]^\theta_k$ that check in fragment θ (the additional inputs ρ 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 $\mathcal{V}_\rho[\![A]\!]^{\mathsf{L}}_k$. The rules TUNBOXVAL and TMVAL can move values from P to 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 $\mathcal{V}_\rho[\![A]\!]^{\mathsf{P}}_k$. These invariants are summarized by a computational type interpretation $\mathcal{C}_\rho[\![A]\!]^\theta_k$, which identifies sets of (non-value) expressions, and is defined mutually with $\mathcal{V}_\rho[\![A]\!]^\theta_k$.

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* [1, 5]. The interpretation is indexed by a number k. Any value v in $\mathcal{V}_\rho[\![A]\!]^{\mathsf{P}}_k$ 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 $\mathcal{V} < \mathcal{C}$.

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 definition, we take a hybrid approach by only counting steps that happen in the P fragment. The difference can be seen by comparing the definitions of $\mathcal{V}_\rho[\![(x : A) \to B]\!]^{\mathsf{L}}_k$ and $\mathcal{V}_\rho[\![(x : A) \to B]\!]^{\mathsf{P}}_k$, which say "$j \leq k$" and "$j < k$" respectively. If all θs in a derivation are L, then no inequalities are strict, so the step-count k never needs to decrease.

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

$$\frac{}{\cdot \models_k \emptyset}\text{ENIL} \qquad \frac{\Gamma \models_k \rho \qquad v \in \mathcal{V}_\rho[\![A]\!]^\theta_k \qquad \Gamma \vdash^{\theta} A : \star}{\Gamma, x :^{\theta} A \models_k \rho[x \mapsto v]}\text{ECONS}$$

Intuitively, $\Gamma \models_k \rho$ asserts that ρ maps term variables to well-behaved values. Because of the premise $\Gamma \vdash^{\theta} A : \star$ it also asserts that Γ 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 [16], the type interpretation would take an input ρ 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 ρ is similar to normalization proofs for systems that have large elimination of datatypes, such as CIC [43].

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, θ and ρ, if $j \leq k$ then $\mathcal{V}_\rho[\![A]\!]^\theta_k \subseteq \mathcal{V}_\rho[\![A]\!]^\theta_j$.*

The next two lemmas are specific to λ^θ because they relate the L and P interpretations of a type. They are used to handle the TSUB and 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, θ and ρ, $\mathcal{V}_\rho[\![A]\!]^{\mathsf{L}}_k \subseteq \mathcal{V}_\rho[\![A]\!]^{\mathsf{P}}_k$ and $\mathcal{C}_\rho[\![A]\!]^{\mathsf{L}}_k \subseteq \mathcal{C}_\rho[\![A]\!]^{\mathsf{P}}_k$.*

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 ρ, if $\mathsf{Mobile}(A)$ then $\mathcal{V}_\rho[\![A]\!]^{\mathsf{P}}_k \subseteq \mathcal{V}_\rho[\![A]\!]^{\mathsf{L}}_k$.*

Finally, for the TCONV rule, we need equal types to have the same interpretation.

LEMMA 7. *Suppose $\rho\, B_1 \Rrightarrow^* A$ and $\rho\, B_2 \Rrightarrow^* A$ and $\Gamma \vdash^{\theta} B_1 : \star$ and $\Gamma \vdash^{\theta} B_2 : \star$ and $\Gamma \models_k \rho$. Then $a \in \mathcal{I}_\rho[\![B_1]\!]^\theta_k$ iff $a \in \mathcal{I}_\rho[\![B_2]\!]^\theta_k$.*

We can now prove soundness by induction on $\Gamma \vdash^{\theta} a : A$. 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 $\Gamma \vdash^{\theta} a : A$ and $\Gamma \models_k \rho$, then $\rho\, a \in \mathcal{C}_\rho[\![A]\!]^\theta_k$.*

COROLLARY 9 (Normalization).
If $\cdot \vdash^{\mathsf{L}} a : A$, then there exists a value v such that $a \leadsto^* v$.

COROLLARY 10 (Soundness of propositional equality).
If $\cdot \vdash^{\mathsf{L}} a : A_1 = A_2$, then there exists some A such that $A_1 \Rrightarrow^* A$ and $A_2 \Rrightarrow^* A$.

$$
\begin{aligned}
\mathcal{V}_\rho[\![\star]\!]_k^\theta &= \{v \mid \cdot \vdash^\theta v : \star\}\\
\mathcal{V}_\rho[\![\mathsf{Nat}]\!]_k^\theta &= \{v \mid \ v \text{ is of the form } \mathsf{S}^n\ \mathsf{Z}\}\\
\mathcal{V}_\rho[\![A@\theta']\!]_k^\theta &= \{v \mid \cdot \vdash^{\theta'} \rho\, A : \star \text{ and } v \in \mathcal{V}_\rho[\![A]\!]_k^{\theta'}\}\\
\mathcal{V}_\rho[\![(x\!:\!A) \to B]\!]_k^{\mathsf{L}} &= \quad \{\lambda x.b \mid \cdot \vdash^{\mathsf{L}} \lambda x.b : \rho\,((x\!:\!A) \to B)\\
&\qquad\quad \text{and } \forall j \le k,\ \text{if } v \in \mathcal{V}_\rho[\![A]\!]_j^{\mathsf{L}} \text{ then } [v/x]b \in \mathcal{C}_{\rho[x\mapsto v]}[\![B]\!]_j^{\mathsf{L}}\}\\
&\quad\ \cup \{\mathsf{ind}\ f\ x.b \mid \cdot \vdash^{\mathsf{L}} \mathsf{ind}\ f\ x.b : \rho\,((x\!:\!A) \to B)\\
&\qquad\quad \text{and } \forall j \le k,\ \text{if } v \in \mathcal{V}_\rho[\![A]\!]_j^{\mathsf{L}} \text{ then } [v/x][\lambda y.\lambda z.(\mathsf{ind}\ f\ x.b)\ y/f]b \in \mathcal{C}_{\rho[x\mapsto v]}[\![B]\!]_j^{\mathsf{L}}\}\\
\mathcal{V}_\rho[\![(x\!:\!A) \to B]\!]_k^{\mathsf{P}} &= \quad \{\lambda x.b \mid \cdot \vdash^{\mathsf{P}} \lambda x.b : \rho\,((x\!:\!A) \to B)\\
&\qquad\quad \text{and } \forall j < k,\ \text{if } v \in \mathcal{V}_\rho[\![A]\!]_j^{\mathsf{P}} \text{ then } [v/x]b \in \mathcal{C}_{\rho[x\mapsto v]}[\![B]\!]_j^{\mathsf{P}}\}\\
&\quad\ \cup \{\mathsf{rec}\ f\ x.b \mid \cdot \vdash^{\mathsf{P}} \mathsf{rec}\ f\ x.b : \rho\,((x\!:\!A) \to B)\\
&\qquad\quad \text{and } \forall j < k,\ \text{if } v \in \mathcal{V}_\rho[\![A]\!]_j^{\mathsf{P}} \text{ then } [v/x][\mathsf{rec}\ f\ x.b/f]b \in \mathcal{C}_{\rho[x\mapsto v]}[\![B]\!]_j^{\mathsf{P}}\}\\
&\quad\ \cup \{\mathsf{ind}\ f\ x.b \mid \cdot \vdash^{\mathsf{P}} \mathsf{ind}\ f\ x.b : \rho\,((x\!:\!A) \to B)\\
&\qquad\quad \text{and } \forall j < k,\ \text{if } v \in \mathcal{V}_\rho[\![A]\!]_j^{\mathsf{P}} \text{ then } [v/x][\mathsf{ind}\ f\ x.b/f]b \in \mathcal{C}_{\rho[x\mapsto v]}[\![B]\!]_j^{\mathsf{P}}\}\\
\mathcal{V}_\rho[\![A+B]\!]_k^\theta &= \quad \{\mathsf{inl}\ v \mid \cdot \vdash^\theta \rho\,(A+B) : \star \text{ and } v \in \mathcal{V}_\rho[\![A]\!]_k^\theta\}\\
&\quad\ \cup \{\mathsf{inr}\ v \mid \cdot \vdash^\theta \rho\,(A+B) : \star \text{ and } v \in \mathcal{V}_\rho[\![B]\!]_k^\theta\}\\
\mathcal{V}_\rho[\![\Sigma x\!:\!A.B]\!]_k^\theta &= \{\langle v_1, v_2\rangle \mid \cdot \vdash^\theta \rho\,(\Sigma x\!:\!A.B) : \star \text{ and } v_1 \in \mathcal{V}_\rho[\![A]\!]_k^\theta \text{ and } v_2 \in \mathcal{V}_{\rho[x\mapsto v_1]}[\![B]\!]_k^\theta\}\\
\mathcal{V}_\rho[\![\mu x.A]\!]_k^{\theta'} &= \{\mathsf{roll}\ v \mid \cdot \vdash^{\theta'} \mathsf{roll}\ v : \rho\,(\mu x.A) \text{ and } \forall j < k, v \in \mathcal{V}_\rho[\![[\mu x.A/x]A]\!]_j^\theta\}\\
\mathcal{V}_\rho[\![a_1 = a_2]\!]_k^\theta &= \{\mathsf{refl} \mid \cdot \vdash^\theta \rho\,(a_1 = a_2) : \star \text{ and } \rho\, a_1 \Rrightarrow^* a \text{ and } \rho\, a_2 \Rrightarrow^* a \text{ for some } a\}\\
\mathcal{V}_\rho[\![A]\!]_k^\theta &= \emptyset \qquad\qquad \text{otherwise}\\
\\
\mathcal{C}_\rho[\![A]\!]_k^{\mathsf{P}} &= \{a \mid \cdot \vdash^{\mathsf{P}} a : \rho\, A \text{ and } \forall j \le k,\ \text{if } a \leadsto^j v \text{ then } v \in \mathcal{V}_\rho[\![A]\!]_{(k-j)}^{\mathsf{P}}\}\\
\mathcal{C}_\rho[\![A]\!]_k^{\mathsf{L}} &= \{a \mid \cdot \vdash^{\mathsf{L}} a : \rho\, A \text{ and } a \leadsto^* v \in \mathcal{V}_\rho[\![A]\!]_k^{\mathsf{L}}\}
\end{aligned}
$$

Figure 7. Type interpretation

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:

$$y :^{\mathsf{L}} \mathsf{Nat} = (\mathsf{Nat} \to \mathsf{Nat}) \vdash^{\mathsf{L}} (\lambda x.x\ x)\ (\lambda x.x\ x) : \mathsf{Nat}$$

This distinguishes λ^θ 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 would get "stuck" on the variable y, since it does not reduce to the appropriate constructor.

The benefit of giving up normalization of open terms is a more generous equality. Since uses of conversion appear in terms in Coq and Agda, they often get in the way of judging two terms which use such conversions equal. In our system, this can not happen. The drawback is that the typechecker can not automatically normalize expressions (since they may diverge), so uses of **`refl`** must be explicit and annotated with a maximum step count. However, in a language with general recursion some explicit proofs are unavoidable, since checking a logical term can involve reducing a programmatic term that appears in its type. Since our language must accommodate such proofs in any case, making conversion unmarked is appealing.

The progress theorem relies on a canonical forms lemma (elided). In the TCONV and TCONTRA cases we need to know that there are no proofs of inconsistent equalities such as $(\mathsf{Nat} \to \mathsf{Nat}) = \mathsf{Nat}$. Therefore, this lemma relies on Corollary 10. The progress theorem is then an easy induction on $\cdot \vdash^\theta a : A$.

Surface language (`Zombie`)

$\Downarrow$ (elaboration)

Annotated language (ZT derivations)

$\Downarrow$ (erasure)

Core language (ZT)

Figure 8. Implementation

THEOREM 11 (Progress). *If $\cdot \vdash^\theta a : A$, then either a is a value, or there exists a' such that $a \leadsto a'$.*

5. Implementation

We have implemented a prototype dependently-typed language, called `Zombie`, based on λ^θ. 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 λ^θ to keep the normalization proof simple. Instead of a single sort $\star$, `Zombie` includes a full predicative hierarchy [22], which allows both polymorphism and type-level functions. We also include a general form for parameterized recursive datatypes, which subsumes Nat, $A + B$, $\Sigma x : A.B$ and $\mu x.A$. Datatypes are always mobile, and `Zombie` provides structural induction for all strictly positive datatypes (not just Nat) following [20]. Finally, Zombie distinguishes between computationally relevant and irrelevant arguments [24], and includes a multiplace conversion operator, called *multiconversion* [34].

Adding these features to λ^θ 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 8. The part of our implementation that most closely resembles λ^θ is the internal language ZT. This language defines the operational behavior of Zombie expressions. However, like λ^θ, 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 previous work [34]. Annotated ZT is a direct representation of ZT typing derivations, marking all uses of conversion, subsumption, cumulativity, and coercion to and from $A@\theta$ 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* [27] 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 @-types by making them implicit. This change complicates the metatheory (see Lemma 2) 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 Sep^3 [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 3.3). To do inductive reasoning, the Sep^3 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 $((\mathsf{rec}\ f\,x.b)\ \ v) = [v/x][\mathsf{rec}\ f\,x.b/f]b$ do not hold with the usual Coq equality because the step counts differ. Conveniently 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 [8], quotient types, and the univalence axiom [42].

Non-constructive fixpoint semantics. The work of Bertot and Komendantsky [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 $A_\perp$, 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 can not 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 $(\overline{A} \to \overline{A}) \to \overline{A}$. 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 Σ-types.

Smith [35] provides an example which shows that Nuprl needs this restriction. Writing $a\downarrow$ for "a terminates", define a Σ-type T of functions which are not total, and recursively define a p which inhabits T.

$$\begin{array}{rcl} \text{Total}\,(f:\mathbb{N}\to\overline{\mathbb{N}}) & \stackrel{\text{def}}{=} & (n:\mathbb{N}) \to (f\ n)\downarrow \\ T & \stackrel{\text{def}}{=} & \Sigma(f:\mathbb{N}\to\overline{\mathbb{N}}).\text{Total}\ f \to \text{False} \\ (p:T) & \stackrel{\text{def}}{=} & \text{fix}\ (\lambda p.\langle g, \lambda h.\text{—}\rangle) \\ g & \stackrel{\text{def}}{=} & \lambda x.\text{if}\ x=0\ \text{then}\ 0\ \text{else}\ \pi_1(p)(x-1) \end{array}$$

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.

λ^θ has almost all the ingredients for this paradox. Instead of a recursively defined pair we can use a recursive function Unit $\to T$, 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 $(\pi_1(p)\downarrow) \to (p\downarrow)$ 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 nontermination to also handle memory effects. Instead of a unary type constructor $\overline{A}$, it adds the indexed type $\{P\}x\!:\!A\{Q\}$ 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 most hold for infinite loops. An equivalent variant [9] is to allow induction on the number of recursive steps an expression takes to normalize.

λ^θ 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 `div_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 Σ-type asserting the property, so that the property is automatically available for recursive calls. This is what we did for `solver` 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, λ^θ 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 TMVAL is similar to the GET rule in ML5, and our $\mathsf{Mobile}\,(A)$ is similar to the judgment A mobile in ML5. On the other hand, unlike λ^θ, 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 λ^θ 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 $\Pi\Sigma$ [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 λ^θ 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 $\leadsto$ can get in the way of equational reasoning. If a is an expression in P there is no way to prove an equation like $(\mathsf{let}\ x = a\ \mathsf{in}\ f\ x) = (f\ a)$, 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 [20]. 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 O, for *oracular*, in addition to L and P. By not allowing O 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, θ, 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 Ott tool [32]. The authors would also like to thank the anonymous reviewers for their considered and helpful comments.

References

[1] Ahmed, A.: Step-indexed syntactic logical relations for recursive and quantified types. In: ESOP '06: European Symposium on Programming. LNCS, vol. 3924. Springer (2006)

[2] Ahn, K.Y., Sheard, T.: A hierarchy of mendler style recursion combinators: taming inductive datatypes with negative occurrences. In: ICFP '11: International Conference on Functional programming. pp. 234–246. ACM (2011)

[3] Ahn, K.Y., Sheard, T., Fiore, M., Pitts, A.M.: The Nax programming language (work in progress) (2012), talk presented at IFL 2012: the 24th Symposium on Implementation and Application of Functional Languages

[4] Altenkirch, T., Danielsson, N.A., Löh, A., Oury, N.: $\Pi\Sigma$: Dependent types without the sugar. Functional and Logic Programming pp. 40–55 (2010)

[5] Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. ACM Trans. Program. Lang. Syst. 23(5), 657–683 (2001)

[6] Augustsson, L.: Cayenne – a language with dependent types. In: ICFP '98: International Conference on Functional Programming. pp. 239–250. ACM (1998)

[7] Barendregt, H.P.: Lambda calculi with types. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science. pp. 117–309. Oxford University Press (1992)

[8] Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. Journal of Functional Programming 13(2), 261–293 (2003)

[9] Bertot, Y., Komendantsky, V.: Fixed point semantics and partial recursion in coq. In: PPDP '08: Principles and practice of declarative programming. pp. 89–96. ACM (2008)

[10] Brady, E.C.: Idris—systems programming meets full dependent types. In: PLPV'11: Programming languages meets program verification. pp. 43–54. ACM (2011)

[11] Capretta, V.: General recursion via coinductive types. Logical Methods in Computer Science 1(2), 1–18 (2005)

[12] Casinghino, C., Sjöberg, V., Weirich, S.: Step-indexed normalization for a language with general recursion. In: MSFP '12: Mathematically Structured Functional Programming. EPTCS, vol. 76, pp. 25–39 (2012)

[13] Chen, C., Xi, H.: Combining programming with theorem proving. In: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming. pp. 66–77. ICFP '05, ACM, New York, NY, USA (2005), `http://doi.acm.org/10.1145/1086365.1086375`

[14] Constable, R.L., Smith, S.F.: Partial objects in constructive type theory. In: Logic in Computer Science (LICS'87). pp. 183–193. IEEE (1987)

[15] Crary, K.: Type Theoretic Methodology for Practical Programming Languages. Ph.D. thesis, Cornell University (1998)

[16] Geuvers, H.: A short and flexible proof of Strong Normalization for the Calculus of Constructions. In: TYPES '94. LNCS, vol. 996, pp. 14–38 (1995)

[17] Girard, J.Y.: Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. Ph.D. thesis, Université Paris VII (1972)

[18] Jia, L., Vaughan, J.A., Mazurak, K., Zhao, J., Zarko, L., Schorr, J., Zdancewic, S.: AURA: A programming language for authorization and audit. In: ICFP '08: International Conference on Functional Programming). pp. 27–38. ACM (2008)

[19] Jia, L., Walker, D.: Modal proofs as distributed programs (extended abstract). In: ESOP'04: European Symposium on Programming. LNCS, vol. 2986, pp. 219–233. Springer (2004)

[20] Kimmell, G., Stump, A., Eades III, H.D., Fu, P., Sheard, T., Weirich, S., Casinghino, C., Sjöberg, V., Collins, N., Ahn, K.Y.: Equational reasoning about programs with general recursion and call-by-value semantics. In: PLPV '12: Programming languages meets program verification. ACM (2012)

[21] Licata, D.R., Harper, R.: Positively dependent types. In: PLPV '09: Programming languages meets program verification. pp. 3–14. ACM (2008)

[22] Luo, Z.: Computation and Reasoning: A Type Theory for Computer Science. Oxford University Press, USA (1994)

[23] McBride, C., McKinna, J.: The view from the left. J. Funct. Program. 14(1), 69–111 (2004)

[24] Miquel, A.: The implicit calculus of constructions - extending pure type systems with an intersection type binder and subtyping. In: TLCA '01: Proceeding of 5th international conference on Typed Lambda Calculi and Applications. LNCS, vol. 2044, pp. 344–359. Springer (2001)

[25] Murphy, VII, T., Crary, K., Harper, R.: Type-safe distributed programming with ML5. In: Trustworthy Global Computing 2007 (2007)

[26] Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: dependent types for imperative programs. In: ICFP '08: International Conference on Functional Programming. pp. 229–240. ACM (2008)

[27] Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. Inf. Comput. 205(4), 557–580 (2007)

[28] Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology (2007)

[29] Peyton-Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: ICFP '06: International Conference on Functional Programming. pp. 50–61. ACM (2006)

[30] Pierce, B.C.: Types and Programming Languages. MIT Press (2002)

[31] Pierce, B.C., Turner, D.N.: Local type inference. In: ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California (1998)

[32] Sewell, P., Nardelli, F., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strnisa, R.: Ott: Effective tool support for the working semanticist. J. Funct. Program. 20(1), 71–122 (2010)

[33] Sheard, T., Linger, N.: Programming in ωmega. In: Horváth, Z., Plasmeijer, R., Soós, A., Zsók, V. (eds.) 2nd Central European Functional Programming School (CEFP). LNCS, vol. 5161, pp. 158–227. Springer (2007)

[34] Sjöberg, V., Casinghino, C., Ahn, K.Y., Collins, N., Eades III, H.D., Fu, P., Kimmell, G., Sheard, T., Stump, A., Weirich, S.: Irrelevance, heterogeneous equality, and call-by-value dependent type systems. In: MSFP '12: Mathematically Structured Functional Programming. EPTCS, vol. 76, pp. 112–162 (2012)

[35] Smith, S.F.: Partial Objects in Type Theory. Ph.D. thesis, Cornell University (1988)

[36] Stump, A., Deters, M., Petcher, A., Schiller, T., Simpson, T.W.: Verified programming in guru. In: Altenkirch, T., Millstein, T.D. (eds.) PLPV. pp. 49–58. ACM (2009)

[37] Svendsen, K., Birkedal, L., Nanevski, A.: Partiality, state and dependent types. In: Typed lambda calculi and applications (TLCA'11). LNCS, vol. 6690, pp. 198–212. Springer (2011)

[38] Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bhargavan, K., Yang, J.: Secure Distributed Programming with Value-dependent Types. In: ICFP '11: International Conference on Functional Programming. pp. 285–296. ACM (2011)

[39] Tait, W.W.: Intensional interpretations of functionals of finite type i. The Journal of Symbolic Logic 32(2), pp. 198–212 (1967)

[40] The Coq Development Team: The Coq Proof Assistant Reference Manual, Version 8.3. INRIA (2010), `http://coq.inria.fr/V8.3/refman/`

[41] The Coq Development Team: The Coq Proof Assistant, Frequently Asked Questions. INRIA (2011), `http://coq.inria.fr/faq/`

[42] The Univalent Foundations Program: Homotopy Type Theory: Univalent Foundations of Mathematics (2013), `http://arxiv.org/abs/1308.0729`

[43] Werner, B.: Une Théorie des Constructions Inductives. Ph.D. thesis, Université Paris 7 (1994)

[44] Winskel, G.: The formal semantics of programming languages: an introduction. MIT Press, Cambridge, MA, USA (1993)