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

There are two current approaches to the design of dependently-typed programming languages. Coq, Epigram, and Agda, which grew out of the logics of proof assistants, require that all expressions terminate. These languages provide decidable type checking and strong correctness guarantees. In contrast, functional programming languages, like Haskell, Dependent ML and Omega, have adapted some features of dependent type theories, but retain a strict division between types and programs. These languages trade termination obligations for more limited correctness assurances.

Here, we combine these two approaches into a single dependently-typed core language. Our goal is to provide a smooth path from functional programming to dependently-typed programming. Unlike traditional dependent type theories and functional languages, this language allows programmers to work with total and partial functions uniformly. The language itself 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.

These two fragments interact in three significant ways. First, there is a subsumption relationship between the fragments. All logical expressions may be used as programs, so the fact that we know stronger properties about an expression does not restrict its use. Second, both fragments internalize consistency classification as a type so that they can safely talk about the values of the other fragment. The logic may soundly reason about effectful, partial functions, and programs may take logical propositions as parameters and require that they be applied to total arguments. Finally, first-order program values, including proofs computed at runtime, may be used as evidence by the logic.

We the language's type safety and termination for its logical fragment. Because proofs may include values produced by programs, this normalization proof combines a standard Girard-Tait method with a step-indexed argument.

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

General Terms Design, Languages, Verification, 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 [4], Dependent ML [32], and Haskell with GADTs [23] treat dependent types as a generalization of ordinary functional programming. Programmers can use the advanced features to give programs more informative types, but any program that would type check in a simple Hindley-Milner type system is valid, including programs that employ general recursion.

On the other hand, languages like Coq [30], Agda [22] and Epigram [19] treat dependently typed programming as a generalization of theorem proving. By using the "pun" that constructive proofs can be read as programs, a theorem prover for constructive logic can be repurposed as a program verification system. However, these systems must disallow nontermination because an infinite loop can be given any type and would therefore make the logic inconsistent.

In this paper we define a core language, λ^θ, which bridges the gap between these two worlds. As such, it can be compared with Capretta's partiality monad [8], which embeds general recursive programs into Coq or Agda, or with Swamy et al.'s language F* [27] which allows general recursion in ordinary programs but also uses the kind-system to track a sublanguage of pure total functions. Our basic setup is similar, although instead of tracking nontermination in types or kinds, we index our typing judgement by a *consistency classifier* θ which may be L ("logic") or P ("program"):

$$\Gamma \vdash^\theta a : A$$

When θ is L we say that a is a proof in the *logical fragment*, which is known to terminate, and A can be read as a valid theorem.

Programmers do not always want to prove that their functions terminate. Some programs, like interpreters and web servers, are meant to run forever. In other cases, reasoning about termination is a distraction—when writing a complicated program the programmer may prefer to spend verification effort on more subtle classes of bugs.

Nontermination can come up in very simple examples. The following function, written in a Haskell-like syntax, correctly computes integer division unless `m` is 0. (The tag `prog` means that this function is defined in the programmatic fragment.)

```
prog div : Nat -> Nat -> Nat
prog div n m = if n < m then 0
                        else 1 + (div (n - m) m)
```

Disappointingly, `div` can not be written directly in Coq or Agda, because it loops when `m` is 0. In order to be attractive to mainstream programmers, our language imposes **no termination-related restrictions on the programmatic fragment**.

1.1 Interactions between the Proofs and Programs

The logical and programmatic fragments can work together in three important ways.[1]

Proofs about programs First, having defined `div` we might wish to verify facts about it, such as division being a left-inverse to multiplication, or less ambitiously that `div 6 2` evaluates to 3. In a dependently-typed language we can state and prove this fact inside the language.

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

The tag `log` above indicates that this definition is in the logical fragment. The proof `refl` is valid when both sides of the equality evaluate to the same result. Disappointingly, in languages like F* or Aura [16] this theorem cannot even be stated, because type-constructors like = cannot be applied to non-value expressions such as `div 6 3`. This example illustrates an important property of our language, called **freedom of speech**: although proofs cannot themselves use general recursion, they are allowed to *refer* to arbitrary programmatic expressions.

Furthermore, lemmas can quantify over program values. For example, consider the following function that joins two partial values, returning the first if they are both present. We want this function to work for arbitrary data, so we define it in the programmatic fragment.

```
prog mplus : Maybe A -> Maybe A -> Maybe A =
  \x. \y. case x of
            Just x  -> x
            Nothing -> y
```

An identity for this function is that if the first value is nothing, then the result is the second. A logical abstraction can safely take a programmatic value as an argument if its type indicates (with `@P`) that it comes from that fragment. The proof of this property is again `refl` because it is determined by the operational behavior of `mplus`.

```
log idl : ((y:(Maybe A)@P)-> mplus Nothing y = y) =
  \y. refl
```

On the other hand, case analysis is required to show the opposite identity. The logical language can safely scrutinize a program value because pattern matching cannot cause divergence.

```
log idr : (x:(Maybe A)@P) -> mplus x Nothing = x) =
  \x. case x of
        Just x1 -> refl
        Nothing -> refl
```

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, imagine writing a complicated SAT-solver that we do not want to prove terminating. In our system, we can write down the type:

```
prog solver : (f:Formula) ->
                 (Sat f + (Sat f->False))@L
```

Since `solver` is written in the programmatic fragment, it may not terminate. The `@L` in its type indicates that if it *does* return a value—either a proof or a disproof of satisfiability—that value was type checked in the logical fragment.

[1] For the sake of intuition, we use several features in these examples that are not part of λ^θ.

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

```
prog f     : Formula
prog isSat : (Sat f + (Sat f -> False))@L = solver f
log  prf   : Prop f  = case isSat of
    inl pfSat -> ...   use proof of satisfiability
    inr pfUnSat -> ... use proof of unsat
```

Furthermore, some program values are available for use by the logic even if they have not been tagged as logical. For example, suppose there is some property `Q` that holds for all lists, which we establish with the lemma `lemQ`.

```
log lemQ : (xs : List Nat) -> Q xs = ...
```

We can prove `Q` for a programmatic list if it has been evaluated.

```
prog xs  : List Nat = compute ()
log  qxs : (Q xs)   = lemQ xs
```

The lemma application is sound because we know, by the type soundness theorem for the programmatic language, that `xs` must actually be a list of numbers. This argument could not cause divergence, so it can be safely used by the logical fragment.

Subsumption Finally, all proofs are programs. Even though a function might be defined in the logical language, it can be passed as an argument when a programmatic value was expected.

For example, suppose we have a logical definition of the boolean `and` function. We can promote it to a programmatic function and use it with arguments whose termination behavior is unknown.

```
prog is_bryllyg : Bool = and (gyre x) (gymble x)
```

The fact that we know more about `and` (i.e. that it terminates) does not restrict how it may be used. We do not need to duplicate its definition among the two fragments.

1.2 Contributions

The λ^θ language incorporates these observations. Concretely, we make the following contributions.

- We define a type system for a simple dependently-typed language. The type system labels the typing judgement to distinguish between total and partial programs (Section 3).
- We give our dependently-typed language a call-by-value operational semantics, with an expressive typing rule for application (Section 3.1).
- We include an equality type that allows uniform reasoning for total and partial expressions. 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 (Section 3.2).
- We introduce a type form $A@\theta$ that internalizes the labelled type judgement. This type can be used by either fragment to manipulate values belonging to the other (Section 3.3).
- We identify a set of "first-order types"—those whose *values* can freely move between the fragments (Section 3.4).
- We include both general recursion and iso-recursive types in the programmatic fragment. These two different sources of nontermination are handled in the same framework (Section 3.6).

Consistency classifiers

$$\theta \quad ::= \quad \mathsf{L} \mid \mathsf{P}$$

Types and Terms

$$\begin{array}{lcl}
a,\ b,\ A,\ B & ::= & \star \mid (x\!:\!A) \to B \mid a = b \mid \mathsf{Nat} \mid A + B \\
& \mid & \mu_\theta x.A \mid A@\theta \\
& \mid & x \mid \lambda x.a \mid \mathsf{rec}\, f\, x.a \mid a\ b \mid \mathsf{refl} \\
& \mid & \mathsf{inl}\, a \mid \mathsf{inr}\, b \\
& \mid & \mathsf{scase}_z\ a\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ y \Rightarrow a_2\} \\
& \mid & \mathsf{Z} \mid \mathsf{S}\ a \mid \mathsf{ncase}_z\ a\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \\
& \mid & \mathsf{roll}\ a \mid \mathsf{unroll}\ a
\end{array}$$

Values

$$\begin{array}{lcl}
v,\ u & ::= & \star \mid (x\!:\!A) \to B \mid a = b \mid \mathsf{Nat} \mid A + B \\
& \mid & \mu_\theta x.A \mid A@\theta \\
& \mid & x \mid \lambda x.a \mid \mathsf{rec}\, f\, x.a \mid \mathsf{refl} \\
& \mid & \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \mathsf{Z} \mid \mathsf{S}\ v \mid \mathsf{roll}\ v
\end{array}$$

Figure 1. Syntax

- We prove that our language is type safe and that the L fragment is normalizing and logically consistent. Our normalization proof uses an interesting combination of traditional and step-indexed logical relations (Section 4).
- We provide a formalization in Coq of our metatheory.[2]

We have previously reported partial versions of the results in this paper in two different workshop papers. In Casinghino, Sjöberg and Weirich [9] we introduced the proof technique of hybrid step-indexed/traditional logical relations, but in the context of a language without dependent types and with explicit term constructors for the @-types. In Sjöberg *et al.* [26] we introduced our typing rules for equality, but in a language which does not enforce termination. This paper combines both of these works into a unified framework, extends it with new functionality and includes a mechanical proof of correctness for the entire system.

2. Syntax and Operational Semantics

The λ^θ language is based on a simple call-by-value, curry-style lambda calculus. Its syntax is shown in Figure 1. Terms and types and the single kind $\star$ (the "type" of types) are drawn from one syntactic category as in pure type systems [5]. 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 language includes recursive functions $\mathsf{rec}\ f\ x.a$, nonrecursive functions $\lambda x.a$, natural numbers (constructed by Z and S a and eliminated by ncase), disjoint unions (constructed by inl a and inr a and eliminated by scase) and iso-recursive types (introduced by roll a and eliminated by unroll a).

Functions in λ^θ can be given dependent types $(x : A) \to B$. In such types, the result type B, can depend on the value x of the argument. This value can appear inside equality types $a = b$, which are assertions that the term a equals the term b. The equality type has a trivial proof refl, which holds when two expressions have the same operational behavior. Equality proofs can be eliminated implicitly, substituting provably equal terms at any point in a typing derivation.

Finally, λ^θ includes the type $A@\theta$, which we discuss in more detail in Section 3.3. Both the introduction and elimination forms for this type are implicit.

[2] Available from `http://www.cis.upenn.edu/~ccasin/lth.tgz`.

$\boxed{a \rightsquigarrow b}$

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

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

$$\frac{}{\mathsf{ncase}_z\ \mathsf{Z}\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \rightsquigarrow [\mathsf{refl}/z]a_1}\ \textsc{SCaseZ}$$

$$\frac{}{\mathsf{ncase}_z\ \mathsf{S}\ v\ \mathsf{of}\ \{Z \Rightarrow a_1; \mathsf{S}\ x \Rightarrow a_2\} \rightsquigarrow [\mathsf{refl}/z][v/x]a_2}\ \textsc{SCaseS}$$

$$\frac{}{\begin{array}{l}\mathsf{scase}_z\ \mathsf{inl}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \\ \quad \rightsquigarrow [\mathsf{refl}/z][v/x]a_1\end{array}}\ \textsc{SCaseInl}$$

$$\frac{}{\begin{array}{l}\mathsf{scase}_z\ \mathsf{inr}\ v\ \mathsf{of}\ \{\mathsf{inl}\ x \Rightarrow a_1; \mathsf{inr}\ x \Rightarrow a_2\} \\ \quad \rightsquigarrow [\mathsf{refl}/z][v/x]a_2\end{array}}\ \textsc{SCaseInr}$$

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

Figure 2. Operational semantics (congruence rules omitted)

Note that expressions do not contain type annotations (i.e. the type system is Curry-style). 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 or are judged with different types. Because type checking depends on showing that terms are equal in a dependent type system, the definition of equality affects the expressivity of the type system.

Due to the lack of annotations, it is not possible to algorithmically compute the type of an λ^θ term. This is not a problem because we do not intend programmers to write these directly. Instead, like in ICC* [6], we expect programmers to use an annotated *surface* language that the type checker elaborates into λ^θ typing derivations.

The key rules of the small-step operational semantics for λ^θ is shown in Figure 2. The semantics is standard, except for two subtleties. First, our definition of values v includes variables. This inclusion is safe because CBV evaluation only substitutes values for variables and it is useful because the λ^θ type checker needs to reason about whether open terms are values.

Second, the forms ncase and scase, bind the variable z inside both branches of the case. The type of this variable asserts an equality between the scrutinee and the pattern of the branch. At runtime, this variable is replaced by refl because the scrutinee must match the pattern for the branch to be taken.

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 3.2 and 3.4, the type system takes advantage of the fact that values cannot induce nontermination. As a result, some typing rules apply only to values. The fact that variables only range over values makes many values available in open terms.

The language defined in this paper is not full-featured enough to write realistic programs, including some of those discussed in the introduction: most conspicuously it lacks polymorphism, type-level computation, user-defined datatypes, and structural recursion in the logical fragment. Adding these features will require more work in the normalization proof, but it should be straightforward.

The goal of the design of λ^θ is to provide a simple setting to study the connection between the logical and programmatic fragments.

3. Type System

$$\boxed{\vdash \Gamma}$$

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

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

$$\frac{(x :^\theta A) \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash^\theta x : A}\textsc{TVar} \qquad \frac{\begin{array}{l}\Gamma, x :^{\mathsf{L}} A \vdash^{\mathsf{L}} b : B \\ \Gamma \vdash^{\mathsf{L}} (x\!:\!A) \to B : \star\end{array}}{\Gamma \vdash^{\mathsf{L}} \lambda x.a : (x\!:\!A) \to B}\textsc{TLam}$$

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

$$\frac{\begin{array}{l}\Gamma \vdash^\theta A : \star \quad \mathsf{FO}\,(A) \\ \Gamma, x :^\theta A \vdash^\theta B : \star\end{array}}{\Gamma \vdash^\theta (x\!:\!A) \to B : \star}\textsc{TArr}$$

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

Figure 3. Typing: contexts, variables and functions

The typing rules are shown in Figures 3–7. As mentioned before, the typing judgment $\Gamma \vdash^\theta a : A$ is indexed by a consistency classifier θ to divide the language into two fragments. The *logical fragment* is restricted to ensure that all terms in this fragment are normalizing. The *programmatic fragment* adds general recursion and recursive types. In the formation rules for contexts and rule TVAR (both in Figure 3) variables in the context Γ are tagged with θ to indicate their fragment. The type of a variable must be valid in the same fragment as its tag (Rule CTYPE).

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 $\Gamma \vdash^\theta A : \star$.

3.1 Functions

There are two ways to define functions in λ^θ. Rule TLAM types non-recursive λ-expressions and can be used in both fragments, whereas rule TREC types recursive rec-expressions, and can only be used in the programmatic fragment. In both of these rules, the variables are introduced into the context with the same θ as is used to check the entire function.

Function types are checked for well-kindedness by the rule TARR. It checks that the domain and range types have sort $\star$. We discuss the third premise $\mathsf{FO}\,(A)$ in Section 3.5.

The rule for function application, TAPP, is a bit different from the usual application rule in pure dependent languages. It has an additional third premise $\Gamma \vdash^\theta [a/x]B : \star$ that checks that the result type is well-formed. We need this premise because our type system includes value restrictions and variables are considered values. Substituting an expression a for the 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. Some previous work [16, 18, 27] 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 an application to an arbitrary argument when there is no dependency.

$$\frac{\begin{array}{l}\Gamma \vdash f : (x : A) \to B \\ \Gamma \vdash v : A\end{array}}{\Gamma \vdash f\ v : [v/x]B} \qquad \frac{\begin{array}{l}\Gamma \vdash f : A \to B \\ \Gamma \vdash a : A\end{array}}{\Gamma \vdash f\ a : B}$$

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 a lack of 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 often 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 partiality monad also places a restriction on function application: the type of the monadic bind operator does not provide a way to propagate type dependencies, so there is no direct way of writing partial functions depending on partial arguments. Stated differently, it is not clear how to fill in the fourth square in this grid.

$$\begin{array}{lcc} & \text{pure} & \text{monadic} \\ \text{simple} & \dfrac{\begin{array}{l}f : A \to B \\ a : A\end{array}}{f\ a : B} & \dfrac{\begin{array}{l}f : A \to M\ B \\ a : M\ A\end{array}}{f =\!\!\ll a : M\ B} \\ \text{dependent} & \dfrac{\begin{array}{l}f : (x\!:\!A) \to B \\ a : A\end{array}}{f\ a : [a/x]B} & ? \end{array}$$

Thus, while it is possible to take an arbitrary (non-terminating) simply-typed program and embed it into Coq or Agda by mechanically changing all constants to uses of monadic return and all function applications to uses of monadic bind, the same transformation does not work for an arbitrary dependently-typed program. The lack of the fourth square means that some nonterminating, but dependently-typed programs are inexpressible. This conflicts with our design goal that there should be no restrictions on the P fragment of our language.

3.2 Equality

One of our design principles is that reasoning about expressions should be based on their run-time behavior, independently of what fragment they are checked in. This principle informs our treatment of equality (Figure 4).

Equality in λ^θ is a primitive type. 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: we can write proofs *about* programs even when those programs are not logical. (Due to subsumption, discussed in Section 3.5, this rule allows us to equate logical and programmatic expressions.)

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 $a \leadsto b$ 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 the language more expressive

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

$$\frac{\Gamma \vdash^{\mathsf{P}} a : A \quad \Gamma \vdash^{\mathsf{P}} b : B}{\Gamma \vdash^{\mathsf{L}} a = b : \star}\textsc{TEq} \qquad \frac{a \Rrightarrow^* c \quad b \Rrightarrow^* c \qquad \Gamma \vdash^{\theta_1} a : A \quad \Gamma \vdash^{\theta_2} b : B}{\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : a = b}\textsc{TRefl} \qquad \frac{\Gamma \vdash^{\mathsf{L}} b : b_1 = b_2 \quad \Gamma \vdash^{\theta} a : [b_1/x]A \qquad \Gamma \vdash^{\theta} [b_2/x]A : \star}{\Gamma \vdash^{\theta} a : [b_2/x]A}\textsc{TConv}$$

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

$$\frac{\Gamma \vdash^{\theta} a : (\mu_{\theta'} x.A) = (\mu_{\theta'} x.B) \qquad \Gamma \vdash^{\theta} ([\mu_{\theta'} x.A/x]A) = ([\mu_{\theta'} x.B/x]B) : \star}{\Gamma \vdash^{\theta} a : ([\mu_{\theta'} x.A/x]A) = ([\mu_{\theta'} x.B/x]B)}\textsc{TMuInv}$$

$$\frac{\Gamma \vdash^{\theta} a : ((x{:}A_1) \to A_2) = ((x{:}B_1) \to B_2) \qquad \Gamma \vdash^{\theta} A_1 = B_1 : \star}{\Gamma \vdash^{\theta} a : A_1 = B_1}\textsc{TArrInv1} \qquad \frac{\Gamma \vdash^{\theta} a : ((x{:}A_1) \to A_2) = ((x{:}B_1) \to B_2) \qquad \Gamma \vdash^{\theta'} v : A_1 \qquad \Gamma \vdash^{\theta} [v/x]A_2 = [v/x]B_2 : \star}{\Gamma \vdash^{\theta} a : [v/x]A_2 = [v/x]B_2}\textsc{TArrInv2}$$

$$\frac{\Gamma \vdash^{\theta} a : (A_1 + A_2) = (B_1 + B_2) \qquad \Gamma \vdash^{\theta} A_1 = B_1 : \star}{\Gamma \vdash^{\theta} a : A_1 = B_1}\textsc{TSumInv1} \qquad \frac{\Gamma \vdash^{\theta} a : (A_1 + A_2) = (B_1 + B_2) \qquad \Gamma \vdash^{\theta} A_2 = B_2 : \star}{\Gamma \vdash^{\theta} a : A_2 = B_2}\textsc{TSumInv2}$$

Figure 4. Typing: equality

and simplifies the proof of preservation. However, because λ^θ includes nontermination, the parallel reduction relation is undecidable. To enable decidable type checking the surface language might include annotations specifying the form of the reduction.

One attractive feature of this approach is that we use the same definition of equality for both fragments. They are both compared "intensionally", i.e. according to the standard operational semantics. By contrast, using a coinductive partiality monad in e.g. Coq, one would compare pure expressions intensionally but define a different, coarser equivalence relation for partial terms which ignores the number of steps they take to normalize. In the coinductive approach, equations like $((\mathsf{rec}\ f\ x.b)\ v) = [v/x][\mathsf{rec}\ f\ x.b/f]b$ do not hold intensionally because the step counts differ.

The fact that our language is CBV is reflected in rule TRefl because $\Rrightarrow$ can only reduce redexes when the active subexpression is a value. We can show $(\lambda x.a)\ v = [v/x]a$ but not the more general $(\lambda x.a)\ b = [b/x]a$. This value restriction reflects the usual equational theory of a CBV language.

However, the choice of CBV has a compensating advantage: because variables only range over values, an equation like $f\ a = x$, where one side is a variable, carries an assertion that $f\ a$ terminates. For example, we can state a theorem like "if division terminates, then it returns a number $\leq$ the input" by the type

```
(i j k : Nat) -> (div i j = k) -> (LE k i)
```

In a theorem proving setting it is crucial to have a way to state that expressions terminate because the principle of induction applies only to total arguments. If we wanted to change λ^θ to be CBN we would add a primitive "terminates" predicate [17].

Equality may be used to convert expressions in types by the elimination rule TConv. Uses of this rule are not marked in the term because they are not relevant at runtime, but again in the surface language, annotations would be necessary. 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. Like in TApp we need to check that b_2 does not violate any value restrictions, so the last premise checks the wellformedness of the type given to the converted term.

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

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

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

$$\frac{\Gamma \vdash^{\theta} v : A@\theta' \qquad \Gamma \vdash^{\theta'} A : \star}{\Gamma \vdash^{\theta'} v : A}\textsc{TUnboxVal}$$

Figure 5. Typing: internalized consistency classification

The rule TContra is used to eliminate 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 constructors* of both sides are defined and unequal, where by head constructor we mean the one of $\star$, $\to$, Nat, $=$, μ, $+$, or $@$. For example, the equation $\mathsf{Nat} = (\mathsf{Nat} \to \mathsf{Nat})$ is considered a contradiction. The six typing rules TAtInv, TMuInv TArrInv1, TArrInv2, TSumInv1, and TSumInv2, make all type constructors injective. Injectivity and discrimination of type constructors is used in our preservation proof, so we discuss these rules further with the metatheory in Section 4.1.

3.3 Internalized Consistency Classification

What is interesting about λ^θ is how its two fragments interact. We manage these interactions through a type that internalizes the typing judgement, which we write as $A@\theta$. 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 disjoint sums of type $A@\mathsf{L} + B$). At the same time, logical lemmas can do the same with values from the programmatic fragment.

The rules for the $A@\theta$ type appear in Figure 5. Both introduction and elimination of this type is unmarked in the syntax. Intuitively, the judgement $\Gamma \vdash^{\theta_1} a : A@\theta_2$ 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).

To get an intuition for these rules, consider a function being applied to an argument from a different fragment:

A general recursive function can require one of its arguments to be a bona-fide proof by marking it @L, i.e. $A@\mathsf{L} \to B$. This type forces that argument to be checked in the logical fragment.

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

In the logical fragment, a logical function can be applied to a programmatic argument by marking it @P. This type forces that argument to be a value checked in the programmatic fragment.

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

This is freedom of speech in action: a logical program handling a programmatic value. Of course, the function f can only be defined in the logical fragment if it is careful to not use its argument in unsafe ways. For example, 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.

Because of the value restriction, 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.[3] However, we can work around this restriction by either first evaluating `a` to a value in the programmatic fragment or by thunking (i.e. we give the function the type $(x : (A \to \mathsf{Unit})@\mathsf{P}) \to B$, pass it $\lambda y.a$ as an argument, and replace x with $x()$ everywhere.)

The @-types are eliminated by the rule TUnboxVal. Again, to preserve termination the rule is restricted to only apply to values. Recall the programmatic function `solver` of type:

```
prog solver : (f:Formula) ->
                  (Sat f + (Sat f->False))@L
```

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

```
prog isSat : (Sat f + (Sat f) -> False)@L = solver f
log  prf = case isSat of
    inl y -> ... here y has logical type Sat f
    inr y -> ... here y has
                 logical type (Sat f -> False)
    ...
```

In this example, the logical program `prf` cannot directly treat `solver f` as a proof (since it may diverge). However, once it has

[3] 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.

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

$$\dfrac{}{\mathsf{FO}\,(\mathsf{Nat})}\,\textsc{FONat} \qquad \dfrac{}{\mathsf{FO}\,(a = b)}\,\textsc{FOEq}$$

$$\dfrac{\mathsf{FO}\,(A) \quad \mathsf{FO}\,(B)}{\mathsf{FO}\,(A + B)}\,\textsc{FOSum} \qquad \dfrac{}{\mathsf{FO}\,(A@\theta)}\,\textsc{FOAt}$$

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

$$\dfrac{\begin{array}{l}\Gamma \vdash^{\mathsf{P}} v : A \quad \mathsf{FO}\,(A)\\ \Gamma \vdash^{\mathsf{L}} A : \star\end{array}}{\Gamma \vdash^{\mathsf{L}} v : A}\,\textsc{TFOVal} \qquad \dfrac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{P}} a : A}\,\textsc{TSub}$$

Figure 6. Typing: first-order types and subsumption

been evaluated to a value, it can be safely used by the logical fragment. Above, let binding forces evaluation of the solver expression, introducing a new programmatic variable `isSat :P (Sat f + (Sat f->False))@L` into the context. Because variables are values, any logical contexts can freely use the variable through TUnboxVal even though it is tagged programmatic.

There are no term constructors corresponding to the introduction and elimination rules for @ in keeping with our philosophy that in a Curry-style system, terms with identical behavior should not be distinguished. Whether the type system deems an expression a to be provably terminating or not does not affect its reduction behavior.

3.4 First-order types

The consistency classifier tracks which expressions are known to come from a normalizing language. For some types of values, however, this distinction is unnecessary. Although the programmatic fragment is logically inconsistent, it is type safe. The standard canonical forms lemma states that types determine the forms of closed values. 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 exactly the same as if it were logical. On the other hand, we can plainly not treat programmatic function values as logical, since they might cause non-termination when applied.

The rule TFOVal exploits this insight by allowing values to be moved from the programmatic to the logical fragment. It relies on an auxiliary judgement $\mathsf{FO}\,(A)$ which identifies the types at which TFOVal may be used. Intuitively, a type is first-order if the same set of values inhabit the type when $\theta = \mathsf{L}$ and when $\theta = \mathsf{P}$. We call these types "first order" to emphasize that they do not include functions.

Concretely, the natural number type Nat is first-order, as is the primitive equality type (which is inhabited by the single constructor refl, as discussed in Section 3.2). Sum types are first-order if their component types are. Finally, @-types are always first-order since they fix a particular θ independent of the one on the typing judgement.

The example in the introduction illustrates how the first-order rule provides useful flexibility for the programmer. The example constructs a programmatic value of type `List Nat` and then uses it in a logical context. The type `List Nat` is a first-order type, because it shares the same values in both the logical and programmatic fragments.

Furthermore, the first-order rule means that first-order types never need to be tagged with logical classifiers. Consider the argument and result types of functions. 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 TFOVAL. Similar, without loss of generality we can use $A \to (a = b)$ instead of $A \to ((a = b)@\mathsf{L})$, since TFOVAL will apply whenever TUNBOXVAL would have. Another consequence is that multiple @'s have no effect beyond the innermost @ in a type. The type $A@\mathsf{P}@\mathsf{L}@\mathsf{P}@\mathsf{L}@\mathsf{P}$ can be used in the same way as the type $A@\mathsf{P}$.

3.5 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 with the same type. Such a rule is useful to avoid code duplication. If a function can be easily defined in the logical fragment one may as well do that, and then use it without penalty in the programmatic fragment. In particular, a logical term can always be supplied to a function expecting a programmatic argument.

However, the inclusion of the TSUB rule forces us to restrict the domains of arrow types to be first-order. We do so with the premise $\mathsf{FO}\,(A)$ in the rule TARR.

What this restriction means in practice is that higher-order functions must use @-types to specify which fragment their arguments belong to. For example, $\mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat}$ is not a well-formed type, so the programmer has to choose either $\mathsf{Nat} \to ((\mathsf{Nat} \to \mathsf{Nat})@\mathsf{L}) \to \mathsf{Nat}$ or $\mathsf{Nat} \to ((\mathsf{Nat} \to \mathsf{Nat})@\mathsf{P}) \to \mathsf{Nat}$. In contrast, first-order arguments do not need nor benefit from tagging.

The reason that function arguments need to be first-order is to account for contravariance. Since proofs are programs, we should be able to introduce a function in the logical fragment and use it in the programmatic:

$$\dfrac{\dfrac{\Gamma, x :^{\mathsf{L}} A \vdash^{\mathsf{L}} b : B}{\Gamma \vdash^{\mathsf{L}} (\lambda x.b) : (x : A) \to B}\ \text{TLAM}}{\Gamma \vdash^{\mathsf{P}} (\lambda x.b) : (x : A) \to B}\ \text{TSUB}$$

Now 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 $\mathsf{FO}\,(A)$ checks.

For this idea to work, it is important that the rules for @-types do not require explicit elimination forms. For example, checking that the type

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

is well-formed implicitly uses TUNBOXVAL. But the equation still talks about the expression `f n`. If we instead had to make the unboxing explicit

```
(f: (Nat->Nat)@P) ->
          (unbox f) (plus n 0) = (unbox f) n
```

then there would be no way to write a logical lemma proving the original equation.

In the previous version of our system [9] the @-elimination forms were explicit. We solved the subsumption issue by requiring that *all* arrow types specify the fragment of their domain in the syntax $(x :^{\theta} A) \to B$. Essentially, this approach fuses TUNBOXVAL and TAPP into a single rule. But it also requires unnecessary clutter in the common case when A is a first-order type and the θ does not matter.

3.6 Data

The final set of typing rules (Figure 7) deals with datatypes: natural numbers, disjoint sums, and—most interestingly—recursive types.

The natural number type Nat and its typing rules (TNAT, TZERO, TSUCC, and TNCASE) are standard. We include it to ensure that there is some base type in the logical language. We allow pattern matching on Nats in the logical fragment.

The rules for sums (TSUM, TINL, TINR, and TSCASE) are mostly standard, with the novelty that TSCASE allows a scrutinee that type checks in one fragment θ' to be eliminated in another fragment θ. The motivation for this added generality is similar to the motivation for first-order types: given a value in $(A_1 + A_2)@\theta'$ we know from the canonical forms lemma that it must be a sum constructor applied to a value from $A_1@\theta'$ or $A_2@\theta'$, so it is safe to scrutinize it. This generality allows the logical language to reason by case analysis on programmatic values. For example, the lemma `idr` from the introduction uses this capability.

Finally, the rules TMU, TROLL and TUNROLL deal with general recursive types. Apart from the θs, these rules are standard for iso-recursive types, i.e. a recursive type and its unfolding are treated as unequal but isomorphic types, and the program contains explicit roll and unroll expressions to map between them. The choice of iso-recursive rather than equi-recursive types is guided by metatheory. The step-indexed logical relation, defined in Section 4.2, uses the step from the reduction $\mathsf{unroll}\,(\mathsf{roll}\,v) \leadsto v$ to show that the definition is well founded.

Recursive types with negative occurrences—that is, with the recursive variable appearing to the left of an arrow, such as $\mu_{\theta} x.(x \to \mathsf{Nat})$—make it possible to write diverging programs using just λ-expressions. They are an additional source of nontermination, independent of explicitly recursive functions. Yet they are useful for programming, e.g. when implementing delimited control [25] or higher-order abstract syntax [24], so we want to allow them in the programmatic fragment.

Wellformedness of recursive types is checked by the rule TMU. We check that A is a wellformed type in a context extended with the variable x. We also tag the type itself by the fragment it was defined in (the θ subscript on the μ) to avoid unsound interactions with subsumption.

In order to ensure normalization, it suffices to only restrict the the elimination rule TUNROLL. The introduction rule TROLL can be used in both fragments. Intuitively, this shows that it is not dangerous to *construct* datatype values; the potential nontermination comes from programs that eliminate them.

This idea that some operations can safely be performed even on negative datatypes seems promising. In future work, we hope to add eliminations of recursive types in the logical fragment. In particular, following the ideas of Ahn and Sheard [2] we hope to add combinators to define recursive functions over recursive data. Excitingly, they show that it is safe to allow function definitions even over negative datatypes, as long as the function unfolds the type "just once". Ultimately we want positive and negative datatypes to co-exist in the same language, defined using the same mechanism but supporting different operations.

4. Metatheory

We now consider the metatheory of λ^{θ}. We are interested in two properties. First, that the entire language is type-safe (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. Preservation can be proved up front. However, since the rules TCONV and TCONTRA allow stuck terms to type check given a contradiction, the progress lemma depends on logical consistency. In our development we first prove preservation, then normalization and consistency, and finally progress.

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

$$\frac{\vdash \Gamma}{\Gamma \vdash^{\mathsf{L}} \mathsf{Nat} : \star}\textsc{TNat} \qquad \frac{\vdash \Gamma}{\Gamma \vdash^{\mathsf{L}} \mathsf{Z} : \mathsf{Nat}}\textsc{TZero} \qquad \frac{\Gamma \vdash^{\theta} a : \mathsf{Nat}}{\Gamma \vdash^{\theta} \mathsf{S}\ a : \mathsf{Nat}}\textsc{TSucc} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\theta} a : \mathsf{Nat}\\ \Gamma, z :^{\mathsf{L}} \mathsf{Z} = a \vdash^{\theta} b_1 : B\\ \Gamma, x :^{\theta} \mathsf{Nat}, z :^{\mathsf{L}} (\mathsf{S}\ x) = a \vdash^{\theta} b_2 : B\\ \Gamma \vdash^{\theta} B : \star\end{array}}{\Gamma \vdash^{\theta} \mathsf{ncase}_z\ a\ \mathsf{of}\ \{Z \Rightarrow b_1; \mathsf{S}\ x \Rightarrow b_2\} : B}\textsc{TNCase}$$

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

$$\frac{\Gamma, x :^{\theta} \star \vdash^{\theta} A : \star}{\Gamma \vdash^{\theta} \mu_{\theta} x.A : \star}\textsc{TMu} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\theta} a : [\mu_{\theta} x.A/x]A\\ \Gamma \vdash^{\theta} \mu_{\theta} x.A : \star\end{array}}{\Gamma \vdash^{\theta} \mathsf{roll}\, a : \mu_{\theta} x.A}\textsc{TRoll} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{P}} a : \mu_{\theta} x.A\\ \Gamma \vdash^{\mathsf{P}} [\mu_{\theta} x.A/x]A : \star\end{array}}{\Gamma \vdash^{\mathsf{P}} \mathsf{unroll}\, a : [\mu_{\theta} x.A/x]A}\textsc{TUnroll}$$

Figure 7. Typing: data

.

The proof of normalization is particularly interesting. In order to handle the rules TFOVal and TUnboxVal, we use a combination of traditional and step-indexed logical relations.

Coq formalization The theorems in this paper have been checked in Coq. To prove certain facts about our logical relation we found it necessary to add a standard axiom of functional extensionality to Coq. This axiom is know to be consistent with Coq's logic [29]. Due to time constraints, the Coq formalization also assumes one lemma about the system—that a step of reduction does not introduce new free variables. This lemma is trivial to prove on paper, but requires substantial arithmetic machinery in Coq. We plan to close this small gap in the immediate future.

4.1 Preservation

The preservation proof as usual relies on weakening, substitution and inversion lemmas. Weakening is completely standard. The substitution lemma mentions the θ' on the variable we are substituting for, and is restricted to substituting values, not arbitrary expressions. This restriction reflects the value restrictions in the rules SBetaFun,SBetaLam, TBoxLV and TUnboxVal, and also the the value restriction in CBV-reduction used by the TRefl rule.

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

So far, these lemmas are not very different from the metatheory of other dependently typed calculi, such as the Calculus of Constructions (CC) [11]. However, our inversion lemmas are more complicated than the corresponding ones for CC. This is because one of the design goals of λ^{θ} is that typing rules without runtime effects should not require term constructors. In particular, uses of TConv and TBoxP/L/LV are not marked in the term. This leaves correspondingly less leverage for inversion lemmas.

As an example, consider inversion for λ-expressions. In CC 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). Further, if the Box rules were used, A may be an @-type. Taking this into account our inversion lemma is:

Lemma 2 (Inversion for λ-expressions). If $\Gamma \vdash^{\theta} (\lambda x.b) : A$, then either

1. $\Gamma \vdash^{\mathsf{L}} p : A = ((x\!:\!B_1) \to B_2)$ and $\Gamma, x :^{\theta} B_1 \vdash^{\theta} b : B_2$, for some expressions p, B_1 and B_2,
2. or $\Gamma \vdash^{\mathsf{L}} p : A = (((x : B_1) \to B_2)@\theta' \ldots @\theta'')$ and $\Gamma, x :^{\theta'} B_1 \vdash^{\theta'} b : B_2$.

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

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

To make this proof go through, we must add type constructor injectivity (rules TArrInv1 etc) and discrimination (TContra) to the language. The problem is due to the weak inversion lemmas. Consider e.g. the case when a function application steps, $(\lambda x.b)\ v \leadsto [v/x]b$. 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 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 $A_1 = B_2$, while in the second case we use TContra.

4.2 Normalization

Our proof of normalization builds upon the standard Girard-Tait reducibility method [14, 28], in a CBV-style formulation. The crux of such a method is to define a "type interpretation". For each type A we define a set of values $[\![A]\!]$. We then prove that the interpretation is "sound": any closed term a of logical type A reduces to a value in $[\![A]\!]$. The definition of the type interpretation is a logical relation and follows the structure of types. For example the usual interpretation of function types $[\![A \to B]\!]$ is the set of λ-expressions that when applied to an argument in $[\![A]\!]$ evaluate to a value in $[\![B]\!]$.

Motivation The proof hinges on the definition of a suitable type interpretation and a suitably generalized statement of soundness.

To motivate the main ideas, we first show two failed proof attempts and how to fix them.

We begin by considering the statement of the soundness theorem. To generalize it to open terms, write ρ for mappings of variables to values, and write $\Gamma \models \rho$ to mean that the values in the mapping are all suitable, i.e. if $(x :^{\theta} A) \in \Gamma$ then $\rho\, x \in \llbracket A \rrbracket$. For now we will be vague about the definition of $\llbracket A \rrbracket$. For normalization, we are ultimately only interested in the L fragment, so our first try is:

Soundness (attempt 1): If $\Gamma \vdash^{\mathsf{L}} a : A$ and $\Gamma \models \rho$, then $\rho\, a \leadsto^* v \in \llbracket A \rrbracket$.

However, if we try to prove this statement by induction on $\Gamma \vdash^{\mathsf{L}} a : A$, we get stuck in the cases for TUNBOXVAL and TFOVAL. We have some value v of which we know nothing except that the it is well-typed in P, and need to show that it belongs to the type interpretation. This difficulty shows that we need to generalize the theorem statement to also say something about programmatic expressions.

Since values can move from L to P, we must also characterize which values are typeable programmatically. We define two different type interpretations $\llbracket A \rrbracket^{\mathsf{L}}$ and $\llbracket A \rrbracket^{\mathsf{P}}$, which should be equal whenever A is a first-order type. For programmatic *expressions* the soundness theorem should express partial correctness: not every programmatic expression terminates, but we want to characterize the ones that do.

Soundness (attempt 2): If $\Gamma \vdash^{\theta} a : A$ and $\Gamma \models \rho$, then

- If θ is L, then $\rho\, a \leadsto^* v \in \llbracket A \rrbracket^{\mathsf{L}}$.
- If θ is P and $\rho\, a \leadsto^* v$, then $v \in \llbracket A \rrbracket^{\mathsf{P}}$.

This idea is correct, but now we have a new problem. We must construct a type interpretation $\llbracket A \rrbracket^{\mathsf{P}}$ that accounts for the rules for recursive functions and recursive types. The construction of logical relations for such language features is a well-known thorny issue. For example, the rule TROLL suggests that the definition of $\llbracket \mu_\theta x.A \rrbracket^{\mathsf{P}}$ should be the set of values $\mathsf{roll}\, v$ such that $v \in \llbracket [\mu_\theta x.A/x]A \rrbracket^{\mathsf{P}}$. Unfortunately, if we define $\llbracket A \rrbracket^{\mathsf{P}}$ by structural recursion on A that is not a valid definition: $[\mu_\theta x.A/x]A$ is a bigger type than $\mu_\theta x.A$.

A well-known solution is to make the interpretation *step-indexed* [1, 3]. The interpretation is given a number k, the *step count*, as an extra argument. Intuitively, a value v is in the interpretation if any use of v will be "well-behaved" for at least k steps of execution. The interpretation is defined by strong induction on the step count.

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.

Proof The definition of our type interpretation is shown in Figure 8. Following Ahmed [1] it is split into two parts: the *value* interpretation $\mathcal{V}_\rho\llbracket A \rrbracket^{\theta}_k$ and the *computational* interpretation $\mathcal{C}_\rho\llbracket A \rrbracket^{\theta}_k$ that mutually refer to each other.

The value interpretation contains closed, well-typed values. We maintain the invariant that if $v \in \mathcal{V}_\rho\llbracket A \rrbracket^{\theta}_k$ then $\cdot \vdash^{\theta} v : \rho\, A$ by adding typing conditions to the definition as appropriate. In addition to the type A, the interpretation also takes a step-count k and a substitution ρ as inputs. The k intuitively indicates that all programmatic subexpressions of the values in $\mathcal{V}_\rho\llbracket A \rrbracket^{\theta}_k$ will be well-behaved for at least k steps of computation.

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

Figure 8. Type interpretation

We only want to count steps in the programmatic fragment. The difference can be seen by comparing the definitions of $\mathcal{V}_\rho\llbracket (x : A) \to B \rrbracket^{\mathsf{L}}_k$ and $\mathcal{V}_\rho\llbracket (x : A) \to B \rrbracket^{\mathsf{P}}_k$, 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 goes down and the interpretation is equivalent to not being step-indexed. We use $j \leq k$ instead of just $j = k$ because the interpretation as a whole needs to be closed under decreasing k (Lemma 4).

The input ρ maps free variables of A to values. The ρ is extended in the interpretation of dependent function types $\mathcal{V}_\rho\llbracket (x : A) \to B \rrbracket^{\theta}_k$. Instead of maintaining ρ as a separate input we could change that case to say "…then $[v/x]a \in \mathcal{C}_\rho\llbracket [v/x]B \rrbracket^{\theta}_j$", but then the definition would not obviously be well founded for logical types.

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. Convertibility by parallel reduction equates enough terms to justify the introduction rule TREFL, and yet is specific enough that we can prove Lemma 7 which justifies the elimination rule TCONV.

In a normalization proof for System F or for CC [13], 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 [31].

The computational interpretation $\mathcal{C}_\rho\llbracket A \rrbracket^{\theta}_k$ contains closed terms. The definition expresses the connection between θ and total/partial correctness. For $\theta = \mathsf{L}$ the term $\rho\, a$ must evaluate to a value. For $\theta = \mathsf{P}$, *if* the term $\rho\, a$ evaluates to a value within k steps, then the value must be in the appropriate interpretation.

The interpretation is defined by well-founded recursion, where the decreasing metric is the lexicographically ordered triple $(k, A, \mathcal{I})$, where k is the step-index, A is the structure of the type, and $\mathcal{I}$ is one of $\mathcal{C}$ or $\mathcal{V}$ with $\mathcal{V} < \mathcal{C}$. The $\mathcal{C}$ interpretation can call the $\mathcal{V}$ interpretation at the same index and type, but not vice versa.

In the proof, we are interested in closing substitutions ρ which map variables to values in the correct interpretation. We inductively define the judgement $\Gamma \models_k \rho$ by

$$\frac{}{\cdot \models_k \emptyset}\text{EWFNIL} \qquad \frac{\begin{array}{c}\Gamma \models_k \rho \\ v \in \mathcal{V}_\rho[\![A]\!]^\theta_k \quad \Gamma \vdash^\theta A : \star\end{array}}{\Gamma, x :^\theta A \models_k \rho[x \mapsto v]}\text{EWFCONS}$$

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.

The soundness theorem relies on a few key lemmas about the interpretation. These are all proven by induction on the same metric that the interpretation is defined by. The first is a standard property for step-indexed logical relations: it says that requiring values to stay well-behaved for a larger number of steps creates a more precise interpretation.

Lemma 4 (Downward closure). 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. The first lemma is exactly what is needed to handle the TFOVAL rule:

Lemma 5. For any k and ρ, if $\mathsf{FO}\,(A)$ then $\mathcal{V}_\rho[\![A]\!]^{\mathsf{L}}_k \subseteq \mathcal{V}_\rho[\![A]\!]^{\mathsf{P}}_k$.

The second lemma is needed to handle the TSUB rule. In the proof of it we use Lemma 5 to deal with contravariance in the case for arrow types.

Lemma 6. For any A, k, θ and ρ, $\mathcal{V}_\rho[\![A]\!]^{\mathsf{L}}_k \subseteq \mathcal{V}_\rho[\![A]\!]^{\mathsf{P}}_k$ and $\mathcal{C}_\rho[\![A]\!]^{\mathsf{L}}_k \subseteq \mathcal{V}_\rho[\![A]\!]^{\mathsf{P}}_k$.

Finally, in order to handle the TCONV rule we need a lemma stating that equal types have the same interpretation.

Lemma 7 (Interpretation respects reduction). 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 $a \in \mathcal{I}_\rho[\![A]\!]^\theta_k$ iff $a \in \mathcal{I}_\rho[\![B]\!]^\theta_k$.

We can now prove soundness by induction on $\Gamma \vdash^\theta a : A$.

Theorem 8 (Soundness of interpretation). If $\Gamma \vdash^\theta a : A$ and $\Gamma \models_k \rho$, then $\rho\, a \in \mathcal{C}_\rho[\![A]\!]^\theta_k$.

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.

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 $A_2 \Rrightarrow^* A$.

4.3 Progress

In order to prove the progress theorem we need a canonical forms lemma. In the TCONV and TCONTRA cases we need to know that there are no proofs of inconsistent equalities such as $(\mathsf{Nat} \to \mathsf{Nat}) = \mathsf{Nat}$. So this lemma relies on Corollary 10.

Lemma 11 (Canonical forms).

- If $\cdot \vdash^\theta v : \mathsf{Nat}@\theta_1 \ldots @\theta_2$ then v is either Z or $\mathsf{S}\ v'$.
- If $\cdot \vdash^\theta v : ((x\!:\!A) \to B)@\theta_1 \ldots @\theta_2$, then v is either $\lambda x.b$ or $\mathsf{rec}\, f\, x.b$.
- If $\cdot \vdash^\theta v : (\mu_\theta x.A)@\theta_1 \ldots @\theta_2$, then v is $\mathsf{roll}\ v'$.
- If $\cdot \vdash^\theta v : (A + B)@\theta_1 \ldots @\theta_2$, then v is $\mathsf{inl}\ v'$ or $\mathsf{inr}\ v'$.

The progress theorem is now an easy induction on $\cdot \vdash^\theta a : A$.

Theorem 12 (Progress). If $\cdot \vdash^\theta a : A$, then either a is a value, or there exists a' such that $a \leadsto a'$.

5. Variations

In this section we explain and justify the design choices of λ^θ.

Explicit introduction and elimination forms First, we have chosen to make the introduction and elimination of the $A@\theta$ type implicit so that typing does not interfere with equality. However, if these forms were explicit, we could make the $A@\theta$ type nonstrict. In such a system, there would be a single typing rule for this construct, with no value restriction when embedding a programmatic term in the logic. Thus:

$$\frac{\Gamma \vdash^\theta a : A}{\Gamma \vdash^{\theta'} \mathsf{box}\, a : A@\theta}$$

Furthermore, the term $\mathsf{box}\, a$ would be a value and freeze the evaluation of a.

However, we choose not to use this semantics because it subverts the CBV nature of λ^θ. Arguments of type $A@\theta$ would be evaluated in a quasi-CBN fashion. There would be no way to ensure that an argument of type $((x\!:\!\mathsf{Nat}) \to \mathsf{Nat})@\mathsf{P}$ actually evaluates to a function value before being passed as the parameter to a function. We already have a mechanism for delaying evaluation in a CBV language—thunking—we do not wish to duplicate it with the @-types.

First-order sums Another alternative that we considered was only allowing the formation sums types $(A_1 + A_2)$ where the two component types are first order. The benefit of this change is that the first premise of the rule TSCASE could be stated more usually as $\Gamma \vdash^\theta a : A_1 + A_2$. When sum types are first order, this premise is equivalent to $\Gamma \vdash^\theta a : (A_1 + A_2)@\theta'$, because if the scrutinee were a value, then it could come from any fragment.

However, restricting sums to first order types seems harsh—in particular, data structures would have to tag all higher-order components with θ. We would like to define data structures that could be used equally well in either fragment, which seems difficult with this restriction.

First-order contexts Requiring sum types to be first-order is one step towards only allowing variables with first-order types in the context. This change would mean that we would not have to tag variable assumptions (contexts would be the more familiar list of bindings $x : A$) and all variables would be valid at any θ.

However, this restriction requires extension in the case of recursive types, which add an assumption $x :^\theta \star$ into the context. Because $\star$ is itself not first-order, we need to use the @-type to tag the appropriate fragment for the recursive type. However, the rules currently do not allow the formation of types like $\star@\theta$—we would have to lift this restriction before requiring first-order contexts. However, this extension is already necessary to add polymorphism. These changes, other than requiring sums to be first-order, do not seem that significant. On the other hand, the benefits also appear minor.

Drop subsumption On the other hand, we could move in the other direction and drop the subsumption rule. This change would have the benefit of removing the FO-restriction on functions. At this time, we do not have enough experience with the language to know whether the additional flexibility in function definition is worth the loss of flexibility from subsumption.

Only logical types It is a property that all types in this system type check both logically and programmatically. In other words, there is no $\Gamma \vdash^{\theta} a : \star$ where θ cannot be L. A simpler language would hard-wire this property in to the type system and force all types to be valid in the logical fragment. That modification would simplify the treatment of recursive types (we would not need the θ) and would allow $\star$ to be a first-order type.

However, we do not want to limit ourselves in this way. We plan to experiment with type system additions that are valid programmatically but not in the logical language. For example, one goal would be to add the axiom $\Gamma \vdash^{\mathsf{P}} \star : \star$ to the programmatic language, but prevent it from being used in the formation of types in the logical language.

6. Related Work

The work most closely related to ours is Capretta's partiality monad [8] and F* [27], and we have compared them to λ^{θ} throughout the paper. To sum up, in the partiality monad there is no direct analog to dependent function application in the programmatic fragment; reasoning about partial programs requires working with a separate notion of equivalence; and it only provides recursive function definitions but not general recursive types. In F* the application rule is restricted to values; there is no way to state equations involving nonterminating expressions; and there is no way to move values from the programmatic to the logical fragment (e.g. a logical case expression cannot destruct a value from the programmatic fragment).

Non-constructive fixpoint semantics Bertot and Komendantsky [7] describe a way to embed general recursive functions into Coq which does not use coinduction. They define a datatype $\mathsf{partial}\ A$ which is isomorphic to the usual $\mathsf{Maybe}\ A$ 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 [10] 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 [12] provides an expressive axiomatization of admissible types, but these conditions lead to significant proof obligations, especially when using Σ-types. Although we have not yet added Σ-types to λ^{θ}, we do not believe that there will be any restrictions on the *programmatic* language similar to admissibility.

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 [15, 21]. In particular, λ^{θ} was inspired by ML5 [21], 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 judgement. Our rule TFOVAL is similar to the GET rule in ML5

$$\frac{\Gamma \vdash^{\theta'} A \quad A\ \mathsf{mobile}}{\Gamma \vdash^{\theta} A}\ \text{GET}$$

The A mobile judgement in ML5 is very similar to our judgement $\mathsf{FO}\,(A)$. On the other hand, unlike λ^{θ}, ML5 does not require that the domain of an arrow type be first-order. As we explained in Section 3.5 we make that restriction to accomodate our rule TSUB, a rule which does not make sense in the context of distributed computation.

Other* TRELLYS *approaches This research was carried out in the context of the TRELLYS project, which aims to develop a new practical dependent programming language. We have been working simultaneously on an alternative design [17], where the logical and programmatic fragments are syntactically separate—in effect rejecting the rule TSUB. One of the gains 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 3.4). In order to do inductive reasoning, the language adds an explicit "terminates" predicate.

7. Future work

We view this system as a framework for future extension. We have structured our language definition and metatheory with the plan of extension. We use a collapsed syntax so that we can add new features, such as polymorphism, type-level computation and large eliminations, with only small change to the language definition. We have formalized our metatheory in Coq so that we can check that future extensions do not invalidate the necessary properties of λ^{θ}.

Irrelevance Dependently-typed programming often requires specificational arguments. These arguments do not affect runtime behavior, so should be irrelevant to an operational-based definition of equivalence. Following Linger and Sheard [20] and Barras and Bernardo [6] we would like to extend this language with such arguments. In our previous workshop paper [26], we discuss the interactions between implicit arguments, programmatic computation, and operational-based equality.

Polymorphism and Type-level computation Although types and terms share a common syntax, this language contains only a single sort. As a result, polymorphic and higher-order types are not supported. We would like to extend both fragments of our language to include these features. Our current plan is to extend the proof following a normalization argument for the Calculus of Constructions [13].

Data structures Currently, only programmatic terms can use recursive datatypes. We have already mentioned the possibility of using Mendler-style combinators following the Nax language of Ahn and Sheard [4]. This language places no restriction on what sorts of data types can be defined or how they can be constructed. Instead, it limits the analysis of data structures to ensure the soundness of the logic. At the same time, we would also like to allow type-level elimination of data structures, i.e. large eliminations. However, again such mechanisms complicate our type interpretation.

Termination case This paper presents the interactions between two different type systems, called L and P. However, we could generalize this framework to allow other sets of typing rules. For example, one reasoning principle that would strengthen the reasoning of the metalogic is *termination-case*—a case analysis on whether a programmatic expression evaluates to a value or diverges [17].

[4] Personal communication

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*. By not allowing O expressions to be used as programs, we could control and track the use of termination case.

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 judgement as a type and observing that some values can move freely allows the fragments to interact in nontrivial ways, thereby 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.

References

[1] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP, 2006*, 2006.

[2] K. Y. Ahn and T. Sheard. A hierarchy of mendler style recursion combinators: taming inductive datatypes with negative occurrences. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 234–246, New York, NY, USA, 2011. ACM.

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

[4] L. Augustsson. Cayenne – a language with dependent types. In *ICFP '98: Proceedings of the 3rd ACM SIGPLAN international conference on Functional Programming*, pages 239–250. ACM, 1998.

[5] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[6] B. Barras and B. Bernardo. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *11th international conference on Foundations of Software Science and Computational Structures (FOSSACS 2008)*, volume 4962 of *LNCS*, pages 365–379. Springer, 2008.

[7] Y. Bertot and V. Komendantsky. Fixed point semantics and partial recursion in coq. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming*, PPDP '08, pages 89–96, New York, NY, USA, 2008. ACM.

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

[9] C. Casinghino, V. Sjöberg, and S. Weirich. Step-indexed normalization for a language with general recursion. In J. Chapman and P. B. Levy, editors, *MSFP '12: Proceedings of the Fourth Workshop on Mathematically Structured Functional Programming*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 25–39. Open Publishing Association, 2012.

[10] R. L. Constable and S. F. Smith. Partial objects in constructive type theory. In *LICS, 1987*, pages 183–193, 1987.

[11] T. Coquand and G. Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[12] K. Crary. *Type Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, 1998.

[13] H. Geuvers. A short and flexible proof of Strong Normalization for the Calculus of Constructions. In *TYPES '94*, volume 996 of *LNCS*, pages 14–38, 1995.

[14] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[15] L. Jia and D. Walker. Modal proofs as distributed programs (extended abstract). In *ESOP, 2004*, pages 219–233. Springer, 2004.

[16] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. AURA: A programming language for authorization and audit. In *ICFP '08:Proceedings of the 13th ACM SIGPLAN international conference on Functional Programming)*, pages 27–38, 2008.

[17] G. Kimmell, A. Stump, H. D. E. III, P. Fu, T. Sheard, S. Weirich, C. Casinghino, V. Sjöberg, N. Collins, and K. Y. Ahn. Equational reasoning about programs with general recursion and call-by-value semantics. In *PLPV '12: Proceedings of the sixth workshop on Programming languages meets program verification*, 2012.

[18] D. R. Licata and R. Harper. Positively dependent types. In *Proceedings of the 3rd workshop on Programming languages meets program verification*, PLPV '09, pages 3–14, New York, NY, USA, 2008. ACM.

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

[20] N. Mishra-Linger and T. Sheard. Erasure and Polymorphism in Pure Type Systems. In *11th international conference on Foundations of Software Science and Computational Structures (FOSSACS 2008)*, volume 4962 of *LNCS*, pages 350–364. Springer, 2008.

[21] T. Murphy, VII, K. Crary, and R. Harper. Type-safe distributed programming with ML5. In *Trustworthy Global Computing 2007*, November 2007.

[22] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

[23] S. Peyton-Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the 11th ACM SIGPLAN international conference on Functional Programming*, pages 50–61, 2006.

[24] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA, 1988. ACM.

[25] C.-C. Shan. A static simulation of dynamic delimited control. *Higher Order Symbol. Comput.*, 20(4):371–401, Dec. 2007. ISSN 1388-3690.

[26] V. Sjöberg, C. Casinghino, K. Y. Ahn, N. Collins, H. D. E. III, P. Fu, G. Kimmell, T. Sheard, A. Stump, and S. Weirich. Irrelevance, heterogeneous equality, and call-by-value dependent type systems. In J. Chapman and P. B. Levy, editors, *MSFP '12: Proceedings of the Fourth Workshop on Mathematically Structured Functional Programming*, volume 76 of *Electronic Proceedings in Theoretical Computer Science*, pages 112–162. Open Publishing Association, 2012.

[27] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure Distributed Programming with Value-dependent Types. In *ICFP '11: Proceedings of the 16th ACM SIGPLAN international conference on Functional Programming*, pages 285–296. ACM, 2011.

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

[29] T. C. D. Team. *The Coq Proof Assistant, Frequently Asked Questions*. INRIA, 2011. URL `http://coq.inria.fr/faq/`.

[30] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.3*. INRIA, 2010. Available from `http://coq.inria.fr/V8.3/refman/`.

[31] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris 7, 1994.

[32] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 214–227, 1999.