Step-Indexed Normalization for a Language with General Recursion

Chris Casinghino Vilhelm Sjöberg Stephanie Weirich
School of Engineering and Applied Science
University of Pennsylvania
ccasin@cis.upenn.edu vilhelm@cis.upenn.edu sweirich@cis.upenn.edu

The TRELLYS project has produced several designs for practical dependently typed languages. These languages are broken into two fragments—a *logical* fragment where every term normalizes and which is consistent when interpreted as a logic, and a *programmatic* fragment with general recursion and other convenient but unsound features. In this paper, we present a small example language in this style. Our design allows the programmer to explicitly mention and pass information between the two fragments. We show that this feature substantially complicates the metatheory and present a new technique, combining the traditional Girard–Tait method with step-indexed logical relations, which we use to show normalization for the logical fragment.

1 Introduction

The TRELLYS project is a collaborative initiative to design a dependently-typed language with simple support for general recursion and other convenient but logically unsound features. To this end, the present authors and their collaborators have proposed languages that are broken into two fragments: a *programmatic* fragment with support for all the desired language features, and a *logical* fragment which can reason about programs but is itself restricted for consistency [6, 31, 18, 29, 30].

As a simple example, consider the following natural number division function written in a Haskell-like syntax:

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

This function computes the integer division of `n` by `m` unless `m` is `0`, in which case it loops forever. We label it "`prog`" to indicate it must be defined in the programmatic fragment described above. Disappointingly, `div` can not be written directly in popular dependently-typed languages like Coq [28] or Agda [24] because it is not total.

There are many sensible properties a programmer might wish to verify about `div`. For example, that `div 6 3` evaluates to `2`, or that `div n m <= n` when `m` is not zero. Even though `div` itself is in the programmatic fragment, we wish to state these properties in the consistent logical fragment. For example:

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

Above, the program (aka proof) `div63` is tagged with "`log`" to indicate that it should be typechecked in the logical fragment. The proof itself is just reflexivity, based on the operational behavior of `div`.

To encourage incremental verification, such a language should also include a way for programs which are not known to be terminating to produce proofs. For example, programmers implementing a

complicated decision procedure might begin by writing in the programmatic fragment and come back to prove termination at a later time. To support passing the proofs produced by such a procedure to the logical fragment, the language may include an *internalized logicality judgement*—programs may assert that other programs typecheck in a certain fragment. We use the new type form $A@\theta$, where θ is L or P for the logical or programmatic fragment, to claim that a term has type A in a particular fragment. For example, a SAT solver which is not known to be terminating might be given the following type:

```
prog solver : (f : Formula) -> Maybe ((Satisfiable f) @ L)
```

Here, `solver` takes in some representation of a formula and optionally produces a proof that it is satisfiable. The type "`(Satisfiable f) @ L`" indicates that if a proof is produced, it will typecheck in the logical fragment, even though the procedure itself is written in the programmatic fragment.

For these internalized judgements to be useful, the language must be able to produce them in one fragment and use them in another. In general, any term which is produced in the logical fragment may be safely used in the programmatic fragment. Additionally, values at certain "first-order" types (including $A@\theta$) may be computed in the programmatic fragment and safely used by the logical fragment.

The metatheory of languages with this collection of features has proved challenging. This paper presents a new technique for demonstrating the normalization (and thus consistency) of the logical fragment in such a language. As we will show, direct adaptations of the Girard–Tait reducibility method [15, 26] are insufficient. Since logical terms are permitted to make use of proofs produced programmatically, it is necessary to simultaneously verify partial correctness properties of the programmatic fragment. To this end, our technique combines the traditional method with step-indexed logical relations [2, 3].

Concretely, our contributions are:

- A small language with an internalized logicality judgement, sum types and recursive types (Section 2). While the language is insufficient for our examples, it retains enough features to exhibit the difficulties we have encountered with traditional proofs (Section 3.2).
- A new, hybrid technique for proving normalization of the language's logical fragment (Section 3.3). This technique combines the Girard–Tait reducibility method with a step-indexed logical relation for simultaneously verifying partial correctness properties of the programmatic fragment. This combination seems to be essential to handle the internalized logicality judgement.
- A formalization of the language's metatheory in Coq, including type safety and normalization (Section 3.5).
- A comparison to related work on dependently-typed languages with general recursion and techniques for reasoning about them (Section 4).

The language we consider in this paper is simply-typed and thus insufficient to represent the examples we have presented so far. However, this smaller language is still complex enough to exhibit the difficulties we have encountered in proving normalization, and we are optimistic that our technique will scale up.

2 Language Definition

The language that we consider in this paper is a variant of the simply-typed call-by-value lambda calculus with recursive types and general recursion. Its syntax is given in Figure 1. The chief novelty is the presence of *consistency classifiers* θ. These classifiers are used by the typing judgement (written $\Gamma \vdash^{\theta} a : A$) to divide the language into two fragments. The *logical fragment*, denoted by L, is a simply-typed

$$
\begin{array}{lrl}
\text{Types} & A,B ::= & \mathsf{Unit} \mid A \stackrel{\theta}{\to} B \mid A + B \mid A@\theta \mid \alpha \mid \mu\,\alpha.A \\
\text{Terms} & a,b ::= & x \mid \mathsf{rec} f\,x.a \mid a\,b \mid \mathsf{box}\,a \mid \mathsf{unbox}\,x = a\,\mathsf{in}\,b \\
 & & \mid () \mid \mathsf{inl}\,a \mid \mathsf{inr}\,a \mid \mathsf{case}\,a\,\mathsf{of}\,\{\mathsf{inl}\,x \Rightarrow a_1; \mathsf{inr}\,x \Rightarrow a_2\} \mid \mathsf{roll}\,a \mid \mathsf{unroll}\,a \\
\text{Language Classifiers} & \theta ::= & \mathsf{L} \mid \mathsf{P} \\
\text{Environments} & \Gamma ::= & \cdot \mid \Gamma, x :^{\theta} A \\
\text{Values} & v ::= & x \mid () \mid \mathsf{inl}\,v \mid \mathsf{inr}\,v \mid \mathsf{rec} f\,x.a \mid \mathsf{box}\,v \mid \mathsf{roll}\,v
\end{array}
$$

Syntactic Abbreviation:

$$\lambda x.a \quad \triangleq \quad \mathsf{rec} f\,x.a \qquad \text{when } f \notin \mathrm{FV}(a)$$

Figure 1: Syntax

lambda calculus with unit and sums. As we will show, all terms in this fragment are normalizing. The *programmatic fragment*, denoted by P, adds general recursion and recursive types. The programmatic fragment is a strict superset of the logical fragment: if $\Gamma \vdash^{\mathsf{L}} a : A$, then $\Gamma \vdash^{\mathsf{P}} a : A$ as well.

Terms in the language may include subexpressions from both fragments. The $\mathsf{box}\,a$ term form and corresponding $A@\theta$ type form mark such transitions. Intuitively, the judgement $\Gamma \vdash^{\theta} \mathsf{box}\,a : A@\theta'$ holds when fragment θ can safely observe that a has type A in the fragment θ'.

2.1 The typing judgement

We now describe the typing rules, given in Figure 2. As shown in rule TVAR, variables in the typing context are tagged with a fragment. When a value is substituted for a variable, the value must check in the corresponding fragment.

The fragments of the language may interact in several ways. Functions have arguments that are tagged with consistency classifiers, as in $A \stackrel{\theta}{\to} B$. The θ here specifies whether the function must be applied to a logical or programmatic term. This classifier does not indicate in which fragment the function itself typechecks, and functions in each fragment are permitted to take arguments from the other. Intuitively, the type may be read as "$A@\theta \to B$", except that users need not explicitly box up arguments to functions. The rules for application (which involve the box form) ensure this does not cause non-termination in the logical fragment, as we will discuss shortly.

There are two rules for type-checking functions. The first, TLAM, checks non-recursive functions in the logical fragment. Here, $\lambda x.b$ is syntax sugar for $\mathsf{rec} f\,x.b$ when f does not occur free in b. The second rule, TREC, checks (potentially) recursive functions in the programmatic fragment. Observe that, in both cases, the consistency classifier for the argument is carried into the context when checking the body, but does not directly influence the classifier of the function itself. The rules are otherwise standard.

The $\mathsf{box}\,a$ form effectively internalizes the typing judgement. It is checked by the three rules, describing the circumstances in which the fragments may safely talk about each other. The first rule, TBOXP, says that the programmatic fragment may internalize any typing judgement—if a has type A in fragment θ, then the programmatic fragment can observe that $\mathsf{box}\,a$ has type $A@\theta$.

Rules TBOXL and TBOXLV check box in the logical fragment and are restricted to ensure termination. The former says that if a itself has type A in the logical fragment, then $\mathsf{box}\,a$ may also be formed in the logical fragment (and checks at type $A@\theta$ for any θ, since logical terms are also programmatic). The latter permits the logical fragment to observe that a term checks programmatically. In that case, the term

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

$$\frac{x :^{\theta} A \in \Gamma}{\Gamma \vdash^{\theta} x : A}\ \textsc{TVar} \qquad \frac{\Gamma, x :^{\theta} A \vdash^{\mathsf{L}} b : B}{\Gamma \vdash^{\mathsf{L}} \lambda x.b : A \,^{\theta}\!\to B}\ \textsc{TLam} \qquad \frac{\Gamma, y :^{\theta} A, f :^{\mathsf{P}} A \,^{\theta}\!\to B \vdash^{\mathsf{P}} a : B}{\Gamma \vdash^{\mathsf{P}} \mathsf{rec} f\, y.a : A \,^{\theta}\!\to B}\ \textsc{TRec}$$

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

$$\frac{\Gamma \vdash^{\theta} a : A@\theta' \quad \Gamma, x :^{\theta'} A \vdash^{\theta} b : B}{\Gamma \vdash^{\theta} \mathsf{unbox} x = a \,\mathsf{in}\, b : B}\ \textsc{TUnbox} \qquad \frac{\Gamma \vdash^{\theta} a : A \,^{\theta'}\!\to B \quad \Gamma \vdash^{\theta} \mathsf{box} b : A@\theta'}{\Gamma \vdash^{\theta} a\, b : B}\ \textsc{TApp} \qquad \frac{}{\Gamma \vdash^{\theta} () : \mathsf{Unit}}\ \textsc{TUnit}$$

$$\frac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{P}} a : A}\ \textsc{TSub} \qquad \frac{\Gamma \vdash^{\mathsf{P}} v : A \quad \mathsf{FO}(A)}{\Gamma \vdash^{\mathsf{L}} v : A}\ \textsc{TFOVal} \qquad \frac{\Gamma \vdash^{\theta} a : A}{\Gamma \vdash^{\theta} \mathsf{inl} a : A + B}\ \textsc{TInl}$$

$$\frac{\Gamma \vdash^{\theta} b : A}{\Gamma \vdash^{\theta} \mathsf{inr} b : A + B}\ \textsc{TInr} \qquad \frac{\Gamma \vdash^{\theta} \mathsf{box} a : (A_1 + A_2)@\theta' \quad \Gamma, x :^{\theta'} A_1 \vdash^{\theta} b_1 : B \quad \Gamma, x :^{\theta'} A_2 \vdash^{\theta} b_2 : B}{\Gamma \vdash^{\theta} \mathsf{case}\ a\ \mathsf{of}\ \{\mathsf{inl}\, x \Rightarrow b_1; \mathsf{inr}\, x \Rightarrow b_2\} : B}\ \textsc{TCase}$$

$$\frac{\Gamma \vdash^{\mathsf{P}} a : [\mu\,\alpha.A/\alpha]A}{\Gamma \vdash^{\mathsf{P}} \mathsf{roll} a : \mu\,\alpha.A}\ \textsc{TRoll} \qquad \frac{\Gamma \vdash^{\mathsf{P}} a : \mu\,\alpha.A}{\Gamma \vdash^{\mathsf{P}} \mathsf{unroll} a : [\mu\,\alpha.A/\alpha]A}\ \textsc{TUnroll}$$

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

$$\frac{}{\mathsf{FO}(\mathsf{Unit})}\ \textsc{FOUnit} \qquad \frac{\mathsf{FO}(A) \quad \mathsf{FO}(B)}{\mathsf{FO}(A + B)}\ \textsc{FOSum} \qquad \frac{}{\mathsf{FO}(A@\theta)}\ \textsc{FOAt}$$

Figure 2: Typing Rules

must be a value to ensure normalization. This restriction still permits the logical fragment to consider programmatic terms (for example, recursive functions are values).

Rule TUNBOX checks the elimination form for boxed terms, which resembles a "let". The term $\mathsf{unbox} x = a \,\mathsf{in}\, b$ typechecks when a has type $A@\theta'$ and b is parameterized by a value of type A in fragment θ'. Intuitively, a will be evaluated first, eventually yielding a value $\mathsf{box} v$, and v is substituted into b. The operational semantics are discussed in more detail below. Note that no additional safety restrictions occur in this rule—the box introduction rules handle everything required to ensure that the logical fragment terminates.

The rule for function application, TAPP, makes use of the infrastructure for internalizing the typing judgement. Recall that function types $A \xrightarrow{\theta} B$ demand arguments from a particular fragment. The box and $A@\theta$ constructs already give us a way to safely check a term in different fragments, so we reuse them here. To check the application $a\,b$ in the fragment θ, we check that a has some function type $A \xrightarrow{\theta'} B$, then check that $\mathsf{box} b$ can be given the type $A@\theta'$ in the current fragment. This has the effect of restricting some applications to programmatic terms in the logical fragment—in general, programmatic arguments to logical functions must be values, ensuring termination.

Rules TUNIT, TINL and TINR deal with the introduction forms for the unit and sum base types. These terms may be used in either fragment and the typing rules are standard. Rule TCASE checks the pattern matching elimination form for sums. Notably, sums that typecheck in one fragment may be eliminated in another—again we use the box infrastructure to ensure that this does not introduce non-termination into the logical fragment.

Two rules describe the relationship between the fragments. As already discussed, any logical term can be used programmatically—this is the content of rule TSUB. Rule TFOVAL is more surprising. It allows potentially dangerous programmatic terms to be used in the logical fragment under certain circumstances. In particular, the term must be a value (to ensure termination) and its type must be "first order". The first-order restriction, formalized by the $\mathsf{FO}(A)$ judgement in the same figure, intuitively means that we move can move *data* but not *computations* from the programmatic fragment to the logical one. For example, moving a natural number computed in P to L is safe, but moving a function from P to L could cause non-termination when the function is applied.

Importantly, $A@\theta$ is a first-order type for any A. The programmatic fragment is permitted to compute logical values, including logical function values, and pass them back to the logical fragment. In a language extended with dependent types, we believe this would be useful for working with proofs. For example, a partial decision procedure could be written in the programmatic fragment and the resulting proofs could be used in the logical fragment if the procedure terminates.

Finally, the language includes iso-recursive types [25]. These are checked by the two rules TROLL and TUNROLL. Recursive types are restricted to the programmatic fragment because they can introduce non-termination.

2.2 Operational Semantics

The language's operational semantics are given in Figure 3. We use standard call-by-value evaluation contexts and a small-step reduction relation. Note that reduction occurs inside $\mathsf{box} a$ terms, motivating some of the restrictions from the previous section. The multi-step reduction relation is indexed by a natural number—this will be useful in the step-indexed logical relation defined in Section 3.

$$\text{Evaluation contexts} \quad \mathscr{E} ::= [\cdot] \mid [\cdot]b \mid v[\cdot] \mid \mathsf{inl}\,[\cdot] \mid \mathsf{inr}\,[\cdot] \mid \mathsf{case}\,[\cdot]\,\mathsf{of}\,\{\mathsf{inl}\,x \Rightarrow a_1; \mathsf{inr}\,x \Rightarrow a_2\} \mid \mathsf{box}\,[\cdot] \mid \mathsf{unbox}\,x = [\cdot]\,\mathsf{in}\,a \mid \mathsf{roll}\,[\cdot] \mid \mathsf{unroll}\,[\cdot]$$

$\boxed{a \rightsquigarrow b}$

$$\frac{a \rightsquigarrow b}{\mathscr{E}[a] \rightsquigarrow \mathscr{E}[b]}\ \textsc{SCtx} \qquad \frac{}{(\mathsf{rec}f\,x.a)\,v \rightsquigarrow [v/x][\mathsf{rec}f\,x.a/f]a}\ \textsc{SBeta}$$

$$\frac{}{\mathsf{case}\,\mathsf{inl}\,v\,\mathsf{of}\,\{\mathsf{inl}\,x \Rightarrow a_1; \mathsf{inr}\,x \Rightarrow a_2\} \rightsquigarrow [v/x]a_1}\ \textsc{SCaseL} \qquad \frac{}{\mathsf{unbox}\,x = \mathsf{box}\,v\,\mathsf{in}\,b \rightsquigarrow [v/x]b}\ \textsc{SUnbox}$$

$$\frac{}{\mathsf{case}\,\mathsf{inr}\,v\,\mathsf{of}\,\{\mathsf{inl}\,x \Rightarrow a_1; \mathsf{inr}\,x \Rightarrow a_2\} \rightsquigarrow [v/x]a_2}\ \textsc{SCaseR} \qquad \frac{}{\mathsf{unroll}\,(\mathsf{roll}\,v) \rightsquigarrow v}\ \textsc{SUnroll}$$

$\boxed{a \rightsquigarrow^n b}$

$$\frac{}{a \rightsquigarrow^0 a}\ \textsc{MSRefl} \qquad \frac{a \rightsquigarrow^k b \quad b \rightsquigarrow b'}{a \rightsquigarrow^{(k+1)} b'}\ \textsc{MSStep}$$

$\boxed{a \rightsquigarrow^* b}$

$$\frac{a \rightsquigarrow^k b}{a \rightsquigarrow^* b}\ \textsc{ASAny}$$

Figure 3: Operational Semantics

3 Metatheory

We now consider the metatheory of the small language presented in Section 2. Of particular interest is the normalization result for the logical fragment, for which we employ a novel combination of traditional and step-indexed logical relations. We motivate and explain this technique in Section 3.3. The system also enjoys standard type-safety properties, as we show in Section 3.1. All the results presented here have been mechanized using the Coq theorem prover, and we briefly describe the formalization in Section 3.5. For this reason, we focus on a high-level description of the techniques and elide most proofs.

3.1 Type Safety

We prove type safety via syntactic progress and preservation theorems [32]. The progress result is direct by induction on typing derivations, using appropriate canonical forms lemmas.

Theorem 1 (Progress). If $\cdot \vdash^\theta a : A$ then either a is a value or $a \rightsquigarrow a'$ for some a'.

For preservation, a substitution lemma is required. Because variables are values and our language includes a value restriction (in the TBoxLV rule), we prove the substitution lemma only for values.

Lemma 2 (Substitution). If $\Gamma, x :^{\theta'} B \vdash^\theta a : A$ and $\Gamma \vdash^{\theta'} v : B$, then $\Gamma \vdash^\theta [v/x]a : A$.

Since we employ a call-by-value operational semantics, this substitution lemma is enough to prove preservation.

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

3.2 Adapting the Girard-Tait Reducibility Method

To motivate the use of step-indexed logical relations in our normalization proof, we will first revisit the standard Girard–Tait reducibility method [15, 26] and examine why more direct adaptations of it fail. Traditional techniques for proving strong normalization typically begin by defining the "interpretation" of each type. That is, for each type A, a set of terms $[\![A]\!]$ is defined approximating the type A and where each term in the set is known to be strongly normalizing. Then a "soundness" theorem is proved, demonstrating that if a has type A then $a \in [\![A]\!]$. This implies a is strongly normalizing.

3.2.1 First attempt: ignoring the programmatic fragment

We begin by modify this technique in two ways to fit our setting. First, since we have a deterministic call-by-value operational semantics, the interpretation of each type will be a set of values (not arbitrary terms). Second, since the terms at a given type differ in the programmatic and logical fragments, we index the interpretation by θ, writing $[\![A]\!]^\theta$.

It is tempting to think that, because we do not care about the normalization behavior of the programmatic fragment, the programmatic interpretation of types can be very simple. Perhaps, for example, just the well-typed values of the appropriate type will do. Consider the following interpretation:

$$
\begin{array}{lcl}
[\![A]\!]^{\mathsf{P}} & = & \{v \mid \cdot \vdash^{\mathsf{P}} v : A\} \\
\\
[\![\mathsf{Unit}]\!]^{\mathsf{L}} & = & \{()\} \\
[\![A+B]\!]^{\mathsf{L}} & = & \{\mathsf{inl}\, v \mid v \in [\![A]\!]^{\mathsf{L}}\} \cup \{\mathsf{inr}\, v \mid v \in [\![B]\!]^{\mathsf{L}}\} \\
[\![A \stackrel{\theta}{\to} B]\!]^{\mathsf{L}} & = & \{\lambda x.a \mid \cdot \vdash^{\mathsf{L}} \lambda x.a : A \stackrel{\theta}{\to} B \text{ and for any } v \in [\![A]\!]^\theta, [v/x]a \leadsto^* v' \in [\![B]\!]^{\mathsf{L}}\} \\
[\![A@\theta]\!]^{\mathsf{L}} & = & \{\mathsf{box}\, v \mid v \in [\![A]\!]^\theta\} \\
[\![\mu\, \alpha.A]\!]^{\mathsf{L}} & = & \emptyset \\
[\![\alpha]\!]^{\mathsf{L}} & = & \emptyset
\end{array}
$$

Here, the logical interpretation of Unit contains only $()$. The logical interpretation of a sum type $A+B$ contains $\mathsf{inl}\, v$ for every v in the interpretation A, and $\mathsf{inr}\, v$ for every v in the interpretation of B. The logical interpretation of functions types is standard, except for the addition of the consistency classifier: $A \stackrel{\theta}{\to} B$ contains the term $\lambda x.a$ if, for any v in the interpretation of the domain, $[v/x]a$ reduces to a value in the interpretation of the range ("related functions take related arguments to related results"). The logical interpretation of $A@\theta$ comprises the values $\mathsf{box}\, v$ where v is in $[\![A]\!]^\theta$. Finally, the logical interpretations of recursive types and type variables are empty, since these are used only in the programmatic fragment.

Before we can state a soundness theorem, we must account for contexts. We use ρ for mappings of variables to terms, and write $\Gamma \models \rho$ if $x :^\theta A \in \Gamma$ implies $\rho x \in [\![A]\!]^\theta$. We let ρa stand for the simultaneous replacement of the variables in a by the corresponding terms in ρ.

In this setting, we would hope to be able to prove the following soundness theorem:

Soundness (take 1): Suppose $\Gamma \vdash^{\mathsf{L}} a : A$ and $\Gamma \models \rho$. Then $\rho a \leadsto^* v \in [\![A]\!]^{\mathsf{L}}$.

In a proof by induction on the typing derivation, most of the cases offer little resistance (the interested reader is encouraged to write out the case for the TLAM and TAPP rules). However, the proof gets stuck at the case for the first order rule:

$$\frac{\Gamma \vdash^{\mathsf{P}} v : A \quad \mathsf{FO}(A)}{\Gamma \vdash^{\mathsf{L}} v : A}\ \text{TFOVAL}$$

Here, we must show that $\rho v \in [\![A]\!]^{\mathsf{L}}$ (substituting values into a value produces a value, so ρv does not step). However, since the premise is in the programmatic fragment, we have no induction hypothesis for v. If $A = \mathsf{Unit}$, we can complete the case using a canonical forms lemma (since we know by a substitution lemma that $\cdot \vdash^{\mathsf{L}} \rho v : \mathsf{Unit}$). However, if A is $B@\mathsf{L}$ we are stuck. We could use a canonical forms lemma to observe that ρv must have the shape $\mathsf{box}\, v'$, but no induction hypothesis for v' is available.

3.2.2 Second attempt: partial correctness for the programmatic fragment

Our previous attempt failed because the language permits values of first-order types to move from the programmatic fragment to the logical fragment, but the theorem we were trying to prove didn't capture any information about the programmatic fragment. To fix this, we might try making two changes. First, the programmatic and logical interpretations should agree at first-order types. Second, the programmatic interpretation and the soundness theorem should be modified to prove a partial correctness result for the programmatic fragment—we'll need to know that *if* a programmatic term normalizes, *then* it is in the appropriate interpretation.

These changes should allow us to handle the previously problematic TFOVAL case. Consider the following modified interpretation, ignoring recursive types for the moment:

$$
\begin{array}{lcl}
[\![\mathsf{Unit}]\!]^{\theta} & = & \{()\} \\
[\![A+B]\!]^{\theta} & = & \{\mathsf{inl}\, v \mid v \in [\![A]\!]^{\theta}\} \cup \{\mathsf{inr}\, v \mid v \in [\![B]\!]^{\theta}\} \\
[\![A\,^{\theta}\!\to B]\!]^{\mathsf{L}} & = & \{\lambda x.a \mid \cdot \vdash^{\mathsf{L}} \lambda x.a : A\,^{\theta}\!\to B \text{ and for any } v \in [\![A]\!]^{\theta}, [v/x]a \leadsto^* v' \in [\![B]\!]^{\mathsf{L}}\} \\
[\![A\,^{\theta}\!\to B]\!]^{\mathsf{P}} & = & \{\mathsf{rec} f\, x.a \mid \cdot \vdash^{\mathsf{P}} \mathsf{rec} f\, x.a : A\,^{\theta}\!\to B \\
 & & \qquad \text{and for any } v \in [\![A]\!]^{\theta}, \text{ if } [v/x][\mathsf{rec} f\, x.a/f]a \leadsto^* v' \text{ then } v' \in [\![B]\!]^{\mathsf{P}}\} \\
[\![A@\theta']\!]^{\theta} & = & \{\mathsf{box}\, v \mid v \in [\![A]\!]^{\theta'}\}
\end{array}
$$

Here, the logical interpretation is unchanged. The programmatic interpretation of the first-order types is now the same as the logical interpretation. Finally, we have modified the programmatic interpretation of function types to state a partial correctness property: *if* a function terminates when passed a value in the interpretation of its domain, *then* the result must be in the interpretation of its range. We now restate the soundness theorem similarly.

Soundness (take 2): Suppose $\Gamma \vdash^{\theta} a : A$ and $\Gamma \models \rho$.

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

With the modified interpretation and soundness theorem, the TFOVAL case now goes through. Because the rule only applies to values, the theorem now yields a useful induction hypothesis for the premise.

Unfortunately, this style of definition introduces a new problem: the programmatic interpretation of recursive types. The previous definition (from Section 3.2.1) is insufficient to handle the TUNROLL case of the new soundness theorem. To extend our partial correctness property, we might demand that when unrolling results in a value, that value is in the interpretation of the unrolled type:

$$
[\![\mu\, \alpha.A]\!]^{\mathsf{P}} \quad = \quad \{\mathsf{roll}\, v \mid \cdot \vdash^{\mathsf{P}} \mathsf{roll}\, v : \mu\, \alpha.A \text{ and } v \in [\![[\mu\, \alpha.A/\alpha]A]\!]^{\mathsf{P}}\}
$$

However, this is not a valid definition. If the interpretation is a function defined by recursion on the structure of types, the substitution in $[\![[\mu\, \alpha.A/\alpha]A]\!]^{\mathsf{P}}$ ruins its well-foundedness.

3.3 A step-indexed interpretation

Happily, a technique exists in the literature to cope with the circularity introduced by iso-recursive types. Step-indexed logical relations [2, 3] add an index to the interpretation, indicating the number of available future execution steps. Terms in the relation are guaranteed to respect the property in question only for the number of steps indicated. The interpretation is defined recursively on this additional index, circumventing the circularity problem we encountered above.

Step-indexed logical relations intuitively describe partial-correctness properties—terms are certified to be well behaved for a finite number of steps. For this reason, they have typically been used to prove safety and program equivalence properties, not normalization. We will adopt a hybrid approach, where the indices track execution of subterms in the programmatic fragment (where we need a partial correctness result) but not in the logical fragment (for which we are proving normalization).

Following Ahmed [2], our interpretation is split into two parts. The *value* interpretation $\mathscr{V}[\![A]\!]^\theta_k$ resembles the interpretations shown in the previous sections. The k index here indicates that when a value appears in a larger term, its programmatic components will be "well-behaved" for at least k steps of computation. The *computational* interpretation $\mathscr{C}[\![A]\!]^\theta_k$ contains closed terms, not just values. Its definition resembles the statement of the soundness theorem from the previous section, with steps counted explicitly. Terms in $\mathscr{C}[\![A]\!]^{\mathsf{L}}_k$ are guaranteed to normalize to values in $\mathscr{V}[\![A]\!]^{\mathsf{L}}_k$. On the other hand, we have a partial correctness property for terms in $\mathscr{C}[\![A]\!]^{\mathsf{P}}_k$—*if* they reach a value in j steps for some $j \leq k$, *then* the value is in $\mathscr{V}[\![A]\!]^{\mathsf{P}}_{k-j}$.

$$
\begin{array}{lcl}
\mathscr{V}[\![\mathsf{Unit}]\!]^\theta_k & = & \{()\} \\
\mathscr{V}[\![A+B]\!]^\theta_k & = & \{\mathsf{inl}\, v \mid v \in \mathscr{V}[\![A]\!]^\theta_k\} \cup \{\mathsf{inr}\, v \mid v \in \mathscr{V}[\![B]\!]^\theta_k\} \\
\mathscr{V}[\![A\,@\,\theta']\!]^\theta_k & = & \{\mathsf{box}\, v \mid v \in \mathscr{V}[\![A]\!]^{\theta'}_k\} \\
\mathscr{V}[\![A^{\,\theta'} \to B]\!]^{\mathsf{L}}_k & = & \{\mathsf{rec} f\, x.a \mid \cdot \vdash^{\mathsf{L}} \mathsf{rec} f\, x.a : A^{\,\theta'} \to B \\
 & & \qquad \text{and } \forall j \leq k, \text{ if } v \in \mathscr{V}[\![A]\!]^{\theta'}_j \text{ then } [v/x]a \in \mathscr{C}[\![B]\!]^{\mathsf{L}}_j\} \\
\mathscr{V}[\![A^{\,\theta'} \to B]\!]^{\mathsf{P}}_k & = & \{\mathsf{rec} f\, x.a \mid \cdot \vdash^{\mathsf{P}} \mathsf{rec} f\, x.a : A^{\,\theta'} \to B \\
 & & \qquad \text{and } \forall j < k, \text{ if } v \in \mathscr{V}[\![A]\!]^{\theta'}_j \text{ then } [v/x][\mathsf{rec} f\, x.a/f]a \in \mathscr{C}[\![B]\!]^{\mathsf{P}}_j\} \\
\mathscr{V}[\![\mu\, \alpha.A]\!]^{\mathsf{L}}_k & = & \emptyset \\
\mathscr{V}[\![\mu\, \alpha.A]\!]^{\mathsf{P}}_k & = & \{\mathsf{roll}\, v \mid \cdot \vdash^{\mathsf{P}} \mathsf{roll}\, v : \mu\, \alpha.A \text{ and } \forall j < k, v \in \mathscr{V}[\![[\mu\, \alpha.A/\alpha]A]\!]^{\mathsf{P}}_j\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathscr{C}[\![A]\!]^{\mathsf{P}}_k & = & \{a \mid \cdot \vdash^{\mathsf{P}} a : A \text{ and } \forall j \leq k, \text{ if } a \rightsquigarrow^j v \text{ then } v \in \mathscr{V}[\![A]\!]^{\mathsf{P}}_{(k-j)}\} \\
\mathscr{C}[\![A]\!]^{\mathsf{L}}_k & = & \{a \mid \cdot \vdash^{\mathsf{L}} a : A \text{ and } a \rightsquigarrow^* v \in \mathscr{V}[\![A]\!]^{\mathsf{L}}_k\}
\end{array}
$$

The value interpretation is similar to the proposed interpretation in the previous section, with two changes. First, the function type cases now refer to the computation interpretation rather than explicitly mentioning the reduction behavior. Second, the step indices track reductions in the programmatic fragment. In particular, note that the programmatic interpretation of function types demands that related functions take related arguments to related results at all *strictly smaller* indices, effectively counting the one beta reduction step that this definition unfolds. The beta step in the logical interpretation is not counted, since we are tracking only the reduction of programmatic components.

Unlike the proposed definition from the previous section, this interpretation is well defined. We can formalize its descending well-founded metric as a lexicographically ordered triple $(k, A, \mathscr{I})$: here, k is the index, A is the type and $\mathscr{I}$ is one of $\mathscr{C}$ or $\mathscr{V}$ with $\mathscr{V} < \mathscr{C}$. The third element of the triple tracks which interpretation is being called—the computational interpretation may call the value interpretation at the same index and type, but not vice-versa.

3.4 Normalization

The step-indexed interpretation from the previous section repairs the problems encountered in the first two proposed interpretations and can be used to prove normalization for the logical fragment. Since our results are formalized in Coq, we give only a high-level overview of the proof here. To begin, we must update the $\Gamma \models \rho$ judgement to account for steps. We now write $\Gamma \models_k \rho$ when $x :^\theta A \in \Gamma$ implies $\rho x \in \mathscr{V}[\![A]\!]_k^\theta$.

Three key lemmas are needed in the main soundness theorem. The first is a standard "downward closure" property that often accompanies step-indexed logical relations. This lemma captures the idea that we build a more precise interpretation of a type by considering terms that must be valid for more steps.

> **Lemma** (Downward Closure): For any A and θ, if $j \leq k$ then $\mathscr{V}[\![A]\!]_k^\theta \subseteq \mathscr{V}[\![A]\!]_j^\theta$ and $\mathscr{C}[\![A]\!]_k^\theta \subseteq \mathscr{C}[\![A]\!]_j^\theta$.

We have two lemmas relating the programmatic and logical interpretations, corresponding to the TFO-VAL and TSUB typing rules. The first says that the two interpretations agree on first-order types:

> **Lemma**: If $\mathsf{FO}(A)$, then $\mathscr{V}[\![A]\!]_k^{\mathsf{L}} = \mathscr{V}[\![A]\!]_k^{\mathsf{P}}$.

The second captures the idea that the logical fragment is a subsystem of the programmatic fragment:

> **Lemma**: For any A and k, $\mathscr{V}[\![A]\!]_k^{\mathsf{L}} \subseteq \mathscr{V}[\![A]\!]_k^{\mathsf{P}}$ and $\mathscr{C}[\![A]\!]_k^{\mathsf{L}} \subseteq \mathscr{C}[\![A]\!]_k^{\mathsf{P}}$.

The content of the soundness theorem is essentially the same as in our second failed attempt, but we can now state it more directly, using the computational interpretation. The theorem is proved by induction on the typing derivation, using the lemmas outlined above.

> **Theorem** (Soundness): If $\Gamma \vdash^\theta a : A$ and $\Gamma \models_k \rho$, then $\rho a \in \mathscr{C}[\![A]\!]_k^\theta$.

The normalization of the logical fragment is a direct consequence of this theorem and the definition of the interpretation.

> **Lemma** (Normalization): If $\cdot \vdash^{\mathsf{L}} a : A$ then there exists a value v such that $a \rightsquigarrow^* v$.

3.5 Formalization

The proof outline above has been formalized with the Coq proof assistant [28]. The proof scripts are written in a heavily automated style, inspired by Chlipala's work on practical dependently typed programming [8, 7]. They are available for download at the first author's website:

> `http://www.seas.upenn.edu/~ccasin/papers/step_normalization.tar.gz`.

The language formalized differs in several minor ways from the one presented in this paper. Namely,

- de Bruijn indices are used for binding instead of explicit names.
- Rather than being syntactic sugar, $\lambda x.a$ is a separate form in the grammar of expressions.
- The reduction relation is formalized with explicit congruence rules rather than evaluation contexts.
- The formalized language includes natural numbers, but not unit.

Additionally, to prove certain facts about the interpretation, we found it necessary to add a standard axiom of functional extensionality to Coq. This axiom is known to be consistent with Coq's logic [27].

4 Related Work

4.1 Step-indexed logical relations

Our proof technique draws heavily from previous work on step-indexed logical relations. The idea to approximate models of programming languages up to a number of remaining execution steps originated in the work of Appel and McAllester on foundational proof-carrying code [3]. They observed that the step indices allowed a natural interpretation of recursive types. Subsequently, Ahmed extended this technique to languages involving impredicative polymorphism, mutable state and other features [2, 1].

Hobor, Dockins and Appel have proposed a general *theory of indirection* which captures many of the common use-cases for step-indexed models [16]. They provide a general framework for applying these approximation techniques to resolve certain types of apparent circularity (similar to the problems with recursive types described above). In a recent draft [14], Dockins and Hobor have used this framework to provide a Hoare logic of total correctness for a small language with function pointers and semantic assertions. This work is closely related to the present development, but with different goals: they prove the soundness of a logic which can reason about termination, while we prove that every term in the logical fragment of our language terminates. We have not yet investigated whether their framework can be adapted to our setting, but this connection is a promising avenue for future work.

4.2 Other approaches to recursion and partiality

Many authors have considered language features to model partiality and recursion in a consistent dependent type theory. The language described in the present paper is much simpler, but our goal is to provide a foundation from which we may scale up to full dependent types, so we compare with some of the most closely related approaches.

Partiality monad Capretta proposed representing potentially non-terminating computations via a coinductive *partiality monad* [5]. This technique can be used in existing languages like Coq and Agda, which already support coinduction [10]. For example, Agda's partiality monad has been used to present subtyping for recursive types [13] and represent potentially infinite parsing trees [12].

There are several differences between these approaches and the one outlined in this paper. Coinduction is a very general method for representing infinite data, which we do not consider. Our approach has the advantage that terminating and potentially partial functions are defined and reasoned about in the same way. By contrast, the reasoning principles for coinductively defined functions in Coq and Agda require the user to consider so-called *guardedness conditions* that are not present for terminating functions. More, we are optimistic that splitting the language into two fragments will allow us to include various other potential sources of logical unsoundness uniformly, restricting them to the programmatic fragment. Admittedly, it remains to be seen how well this will work in practice and whether our proof technique will scale.

Partial Types Constable and Smith [9] proposed adding partiality to the Nuprl type theory through the addition of a type $\overline{A}$ of potentially nonterminating computations of type A. The general fixpoint operator, for defining recursive computations then has type

$$(\overline{A} \to \overline{A}) \to \overline{A}.$$

However, to preserve the consistency of the logic in dependent type theories, the type A must be restricted to *admissible*. types. Crary [11] provides an expressive axiomatization of admissible types, but the resulting conditions lead to significant proof obligations, in particular when using Σ types. Although we have not yet formally proven the soundness of the system with arbitrary dependent types (including Σ types), we do not believe that there will be any restrictions on the *programmatic* language, similar to admissibility.

The "later" modality Nakano [23] introduced the "later" modality to define a total language with guarded recursive types. Intuitively, a term of type $\bullet A$ (pronounced "later A") will be useable as a term of type A in the future. The recursive type $\mu\,\alpha.A$ then unfolds to $[\bullet\mu\,\alpha.A/\alpha]A$ rather than $[\mu\,\alpha.A/\alpha]A$. Using this modality, he is able to give the type $(\bullet A \to A) \to A$ to the Y combinator. This type allows programmers to define a variety of recursive functions while still ensuring that the language is normalizing. Nakano uses a step-indexed realizability interpretation to prove the normalization property for his langauge, suggesting deep connections with the present work. One substantial difference is that Nakano's calculus is not call-by-value.

The later modality has been used by subsequent authors to design langauges for a variety of purposes. Krishnaswami and Benton use it define a total language for functional reactive programming [20, 19]. Birkedal et al. [4] study the topos of trees, which they observe can model an extension of Nakano's calculus to a full dependent type theory with guarded recursion. While these authors do not consider languages with partiality and their settings have substantial differences from our own, their success in extending step-indexing and closely-related techniques to model recursion in larger languages is promising.

Other TRELLYS approaches The TRELLYS group has been working simultaneously on an alternative design, where the logical and programmatic languages are completely separate at a syntactic level [18]. This considerably simplifies the metatheory for the logical language, which is no longer a general programming language but rather a collection of principles for reasoning about the programmatic language. On the other hand, it can restrict the expressiveness of the logic and create duplication between the two fragments. We are exploring these trade-offs in our ongoing research.

4.3 Modal type systems for distributed computation

Modal logics allow one to reason from multiple perspectives, called "possible worlds". It is tempting to view the language presented here as such a system, where the possible worlds are θ, the logical and computational fragments of the language.

One way to define a modal logic is to make the world explicit, for example using a judgement $\Gamma \vdash^{\theta} A$, stating that under the assumptions in Γ, the proposition A is true at the world θ. Each assumption in the context is tagged with the world where it holds (θ, A).

$$\frac{(\theta, A) \in \Gamma}{\Gamma \vdash^{\theta} A}$$

In such as system, the `at` modality [17], internalizes the typing judgment into a proposition, with introduction form

$$\frac{\Gamma \vdash^{\theta'} A}{\Gamma \vdash^{\theta} A@\theta'}$$

and elimination form:

$$\frac{\Gamma \vdash^{\theta} A@\theta' \qquad \Gamma, (\theta', A) \vdash^{\theta} C}{\Gamma \vdash^{\theta} C}$$

Our first-order rule is similar to the perspective-shifting rule, called **get**, from ML5 [22, 21].

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

This rule, shown above, allows a class of propositions to be directly translated between worlds. The class of mobile types is very similar to our class of first-order types. For example, base types (such as strings and integers) and the `at` modality ($A@\theta$) are always mobile, sums ($A+B$) are mobile only when their components (A and B) are mobile, but implications are never mobile.

One difference between our system and modal logics is our treatment of implication (i.e. function types). The functions in our system are annotated with a domain fragment, but this is not typically the case in modal logics, where the domain and range of implications are in the same world. Such an approach is incompatible with our subsumption rule:

$$\frac{\Gamma \vdash^{\mathsf{L}} a : A}{\Gamma \vdash^{\mathsf{P}} a : A} \quad \textsc{TSub}$$

Suppose A were a function type $B_1 \to B_2$ with no tag on the domain. When we defined such a function in the logical fragment, the function's body could make use of the fact that its argument checks logically. If the subsumption rule were used to transport the function to the programmatic fragment, it could be applied to terms that check only programmatically, potentially violating assumptions of its body.

5 Conclusion

In this paper, we have presented a small language with two fragments. The *programmatic* fragment supports general recursion and recursive types, while every term in the *logical* fragment is normalizing. Despite these differences, each fragment may explicitly mention and manipulate terms from the other using the *internalized logicality type*, $A@\theta$. We showed that direct adaptations of the Girard–Tait reducibility method fail to yield a normalization proof for the logical fragment. Finally, we proposed a new technique involving step-indexed logical relations and used it to complete the proof, which has been formalized in Coq.

The language considered here is small and unsuitable for real programming tasks. However, it constitutes the core of one our designs for the TRELLYS programming language, and the metatheoretic difficulties we explained and solved in this paper also appear there. In future work, we plan to add polymorphism, type-level computation and dependent types back to this system. If our proof technique scales, this will provide the basis for a practical, dependently-typed programming language which can naturally express and reason about non-terminating computations.

References

[1] Amal Ahmed (2004): *Semantics of Types for Mutable State*. Ph.D. thesis, Princeton University. Available at `http://www.cs.princeton.edu/research/techreps/TR-713-04`.

[2] Amal Ahmed (2006): *Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types*. In: *ESOP, 2006*, doi:10.1007/11693024_6.

[3] Andrew W. Appel & David A. McAllester (2001): *An indexed model of recursive types for foundational proof-carrying code*. *ACM Trans. Program. Lang. Syst.* 23(5), pp. 657–683, doi:10.1145/504709.504712.

[4] Lars Birkedal, Rasmus Ejlers Mogelberg, Jan Schwinghammer & Kristian Stovring (2011): *First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees*. In: *LICS 2011*, pp. 55–64, doi:10.1109/LICS.2011.16.

[5] Venanzio Capretta (2005): *General Recursion via Coinductive Types*. *Logical Methods in Computer Science* 1(2), pp. 1–18, doi:10.2168/LMCS-1(2:1)2005.

[6] Chris Casinghino, Harley D. Eades III, Garrin Kimmell, Vilhelm Sjöberg, Tim Sheard, Aaron Stump & Stephanie Weirich: *The Preliminary Design of the Trellys Core Language*. Available at `http://www.seas.upenn.edu/~ccasin/papers/plpv11_slides.pdf`. Talk and discussion session at PLPV 2011.

[7] Adam Chlipala (2010): *An Introduction to Programming and Proving with Dependent Types in Coq*. *Journal of Formalized Reasoning* 3(2), pp. 1–93. Available at `http://adam.chlipala.net/papers/CpdtJFR/`.

[8] Adam Chlipala (2011): *Certified Programming with Dependent Types*. Available at `http://adam.chlipala.net/cpdt/`.

[9] Robert L. Constable & Scott Fraser Smith (1987): *Partial Objects in Constructive Type Theory*. In: *LICS, 1987*, pp. 183–193.

[10] Thierry Coquand (1994): *Infinite objects in type theory*. In: *Proceedings of the international workshop on Types for proofs and programs*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, pp. 62–78. Available at `http://dl.acm.org/citation.cfm?id=189973.189976`.

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

[12] Nils Anders Danielsson (2010): *Total parser combinators*. In: *ICFP, 2010*, ACM, New York, NY, USA, pp. 285–296, doi:10.1145/1863543.1863585.

[13] Nils Anders Danielsson & Thorsten Altenkirch (2010): *Subtyping, Declaratively*. In Claude Bolduc, Jules Desharnais & Bchir Ktari, editors: *Mathematics of Program Construction*, *Lecture Notes in Computer Science* 6120, Springer Berlin / Heidelberg, pp. 100–118, doi:10.1007/978-3-642-13321-3_8.

[14] Robert Dockins & Aquinas Hobor (2010): *A Theory of Termination via Indirection*. In Amal Ahmed, Nick Benton, Lars Birkedal & Martin Hofmann, editors: *Modelling, Controlling and Reasoning About State*, *Dagstuhl Seminar Proceedings* 10351, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany. Available at `http://drops.dagstuhl.de/opus/volltexte/2010/2805`.

[15] Jean-Yves Girard (1972): *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. thesis, Université Paris VII.

[16] Aquinas Hobor, Robert Dockins & Andrew W. Appel (2010): *A Theory of Indirection via Approximation*. In: *POPL, 2010*, pp. 171–185. Available at `http://msl.cs.princeton.edu/indirection.pdf`.

[17] Limin Jia & David Walker (2004): *Modal proofs as distributed programs (Extended Abstract)*. In: *ESOP, 2004*, Springer, pp. 219–233, doi:10.1007/978-3-540-24725-8_16.

[18] Garrin Kimmell, Aaron Stump, Harley D. Eades III, Peng Fu, Tim Sheard, Stephanie Weirich, Chris Casinghino, Vilhelm Sjöberg, Nathan Collins & Ki Yung Ahn (2012): *Equational Reasoning about Programs with General Recursion and Call-by-value Semantics*. In: *PLPV, 2012*.

[19] Neelakantan Krishnaswami & Nick Benton (2011): *A semantic model for graphical user interfaces*. In: *ICFP, 2011*, ACM, New York, NY, USA, pp. 45–57, doi:10.1145/2034773.2034782.

[20] Neelakantan Krishnaswami & Nick Benton (2011): *Ultrametric Semantics of Reactive Programs*. In: *LICS, 2011*, pp. 257–266, doi:10.1109/LICS.2011.38.

[21] Tom Murphy, VII (2008): *Modal Types for Mobile Code*. Ph.D. thesis, Carnegie Mellon. Available at `http://tom7.org/papers/`. Available as technical report CMU-CS-08-126.

[22] Tom Murphy, VII, Karl Crary & Robert Harper (2007): *Type-safe Distributed Programming with ML5*. In: *Trustworthy Global Computing 2007*, doi:10.1007/978-3-540-78663-4_9.

[23] Hiroshi Nakano (2000): *A Modality for Recursion*. In: *LICS, 2000*, IEEE Computer Society, Washington, DC, USA, pp. 255–, doi:10.1109/LICS.2000.855774.

[24] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory*. Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology.

[25] Benjamin C. Pierce (2002): *Types and Programming Languages*. MIT Press.

[26] W. W. Tait (1967): *Intensional Interpretations of Functionals of Finite Type I*. The Journal of Symbolic Logic 32(2), pp. pp. 198–212, doi:10.2307/2271658.

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

[28] The Coq Development Team (2011): *The Coq Proof Assistant Reference Manual, Version 8.3*. INRIA. Available at `http://coq.inria.fr/V8.3/refman/`.

[29] Stephanie Weirich (2011): *Combining Proofs and Programs*. Invited lecture for RTA 2011 and TLCA 2011, Novi Sad, Serbia.

[30] Stephanie Weirich (2011): *Combining Proofs and Programs*. Presentation at DTP 2011, Shonan Meeting Seminar 007, Japan.

[31] Stephanie Weirich (2011): *Combining Proofs and Programs in Trellys*. Plenary Address at MFPS 26, Pittburgh, PA.

[32] Andrew K. Wright & Matthias Felleisen (1992): *A Syntactic Approach to Type Soundness*. Information and Computation 115, pp. 38–94.