Termination Casts: A Flexible Approach to Termination with General Recursion

Aaron Stump
Computer Science
The University of Iowa
astump@acm.org, Vilhelm Sjöberg
Computer and Information Science
University of Pennsylvania
vilhelm@cis.upenn.edu and Stephanie Weirich
Computer and Information Science
University of Pennsylvania
sweirich@cis.upenn.edu

Abstract

This paper proposes a type-and-effect system called $\mathtt{T}^{\mathtt{eq}\downarrow}$, which distinguishes terminating terms and total functions from possibly diverging terms and partial functions, for a lambda calculus with general recursion and equality types. The central idea is to include a primitive type-form "Terminates t", expressing that term t is terminating; and then allow terms t to be coerced from possibly diverging to total, using a proof of Terminates t. We call such coercions *termination casts*, and show how to implement terminating recursion using them. For the meta-theory of the system, we describe a translation from $\mathtt{T}^{\mathtt{eq}\downarrow}$ to a logical theory of termination for general recursive, simply typed functions. Every typing judgment of $\mathtt{T}^{\mathtt{eq}\downarrow}$ is translated to a theorem expressing the appropriate termination property of the computational part of the $\mathtt{T}^{\mathtt{eq}\downarrow}$ term.

1 Introduction

Soundly combining general recursion and dependent types is a significant current challenge in the design of dependently typed programming languages. The two main difficulties raised by this combination are (1) type-equivalence checking with dependent types usually depends on term reduction, which may fail to terminate in the presence of general recursion; and (2) under the Curry-Howard isomorphism, non-terminating recursions are interpreted as unsound inductive proofs, and hence we lose soundness of the type system as a logic.

Problem (1) can be addressed simply by bounding the number of steps of reduction that can be performed in a single conversion. This solution may seem ad hoc, but it is less problematic if one works, as we do here, with a primitive notion of propositional equality, and no automatic conversion. Explicit casts with equality proofs are used to change the types of terms, and so with a bound on the number of reduction steps allowed, one may simply chain together a sequence of conversions to accommodate long-running terms in types. There are certainly some issues to be addressed in making such a solution workable in practice, but it is not a fundamental problem.

Problem (2), on the other hand, cannot be so easily dealt with, since we must truly know that a recursive function is total if we are to view it soundly as an inductive proof. One well-known approach to this problem was proposed by Capretta [7]: extend a terminating type theory (that is, one for which we have a sound static analysis for totality, which we use to require all functions to be total) with general recursion via coinductive types. Corecursion is used to model general-recursive functions, without losing logical soundness: productive corecursive functions correspond to sound coinductive arguments. The type constructor $(\cdot)^{\nu}$ for possibly diverging computations, together with natural operations on it, is shown to form a monad.

A separate problem related to (2) is extending the flexibility of totality checking for total type theories. It is well-known that structural termination can become awkward for some functions like, for example, natural-number division, where a recursive call must be made on the result of another function call. For this situation, methods like type-based termination have been proposed: see Barthe et al. [4] and several subsequent works by those authors; also, Abel [1]. The idea in type-based termination is, roughly, to associate sizes with data, and track sizes statically across function calls. Recursive calls must be on data with smaller size. This method certainly increases the range of functions judged total in their natural presentation. No static termination analysis will be complete, so there will always be programs that type-based termination cannot judge terminating. When such analyses fail, programmers must rewrite their code so that its termination behavior is more apparent to the analysis. What is required is a flexible method for such explicit termination arguments.

This paper's contribution This paper proposes a system called $\mathtt{T}^{\mathtt{eq}\downarrow}$ that can be seen as building on both these lines of work. We develop a type-and-effect system where the effect distinguishes total from possibly partial terms. The type assignment judgment $\Gamma \vdash t : T\ \theta$ includes a *termination effect* θ, which can be either $\downarrow$ (called "total"), for terms that are known to terminate, or $?$ (called "general"), for terms whose termination behavior is unknown.

We can view this approach as building, at least in spirit, on Capretta's approach with the partiality monad, thanks to the close connection between monads and effects, as shown by Wadler and Thiemann [19]. Of course, there are important differences between the monadic and effectful approaches, most notably that effects are hard-wired into the language definition, while monads are usually programmer-defined. We adopt the effectful approach here, since we are particularly focused on these two kinds of computation, terminating and possibly partial, as fundamental. We thus deem them appropriate for hard-wiring into the language itself. Exploring the tradeoffs more deeply between these two approaches must remain to future work.

Importantly, $\mathtt{T}^{\mathtt{eq}\downarrow}$ provides a flexible approach to termination because the judgment of totality, $\Gamma \vdash t : T\ \downarrow$, is internalized into the type system. The type $\mathbf{Terminates}\ t$ expresses termination of term t. The effect of a term can thus be changed from possibly partial to total by casting the term t with a proof of $\mathbf{Terminates}\ t$. These *termination casts* change the type checker's view of the termination behavior of a term, much as a (sound) type cast changes its view of the type of the term. Termination casts are used with the terminating recursion operator: the body of the putatively terminating recursive function is type-checked under the additional explicit assumption that calls with a structurally smaller argument are terminating.

By reifying this basic view of structural termination as an explicit typing assumption, we follow the spirit of type-based termination: our method eliminates the need for a separate structural check (proposed as an important motivation for type-based termination [4]), and gives the programmer even more flexibility in the kind of functions s/he can write. This is because instead of relying on a static analysis to track sizes of datatypes, our approach allows the user (or an automated reasoning system) to perform arbitrarily complex reasoning to show termination of the function. This reasoning can be internal, using termination casts, or completely external: one can write a general-recursive function that the type checker can only judge to be possibly partial, and later prove a theorem explicitly showing that the function is terminating. Of course, one could also wish to support what we would see as a hybrid approach, in the style of the PROGRAM tactic in Coq [16], but this is outside the scope of the present paper.

Outline of the development In Section 2, we first present the syntax, reduction rules and type assignment system for $\mathtt{T}^{\mathtt{eq}\downarrow}$. Because type assignment is not algorithmic for $\mathtt{T}^{\mathtt{eq}\downarrow}$, we also develop an annotated version of $\mathtt{T}^{\mathtt{eq}\downarrow}$ suitable for implementation, where terms are annotated to enable algorithmic type checking. We follow this explanation with a number of examples of the use of termination casts, in Section 3. Next, in Section 4 we develop our central meta-theoretic result, based on a translation of $\mathtt{T}^{\mathtt{eq}\downarrow}$ typing judgments to judgments about termination of the term in question, formulated in

$$
\begin{array}{llll}
\mathit{effects} & \theta, \rho & ::= & \downarrow \mid ? \\
\mathit{types} & T & ::= & \mathbf{nat} \mid \Pi^{\theta} x : T . T' \mid t = t' \mid \mathbf{Terminates}\ t \\
\mathit{terms} & t & ::= & x \mid \lambda\, x \,.\, t \mid t\, t' \mid 0 \mid \mathbf{Suc}\, t \\
 & & \mid & \mathbf{rec}\, f(x) = t \mid \mathbf{case}\ t\ t'\ t'' \\
 & & \mid & \mathbf{join} \mid \mathbf{terminates} \mid \mathbf{contra} \mid \mathbf{abort} \\
\mathit{values} & v & ::= & x \mid 0 \mid \mathbf{Suc}\, v \mid \lambda\, x \,.\, t \mid \mathbf{rec}\, f(x) = t \\
 & & \mid & \mathbf{join} \mid \mathbf{terminates} \mid \mathbf{contra} \\
\mathit{contexts} & \mathcal{C} & ::= & [] \mid \mathbf{Suc}\, \mathcal{C} \mid \mathcal{C}\, t \mid v\, \mathcal{C} \mid \mathbf{case}\, \mathcal{C}\, t\, t
\end{array}
$$

Figure 1: Syntax of $\mathtt{T}^{\mathtt{eq}\downarrow}$

a first-order logical theory of general-recursive functions (called W'). This system is similar in spirit to Feferman's theory W (see Chapter 13 of [10]), although with significant syntactic differences, and support for hypothetical reasoning about termination. We show that $\mathtt{T}^{\mathtt{eq}\downarrow}$ is sound with respect to this translation. Also, we find that constructive reasoning suffices for soundness of the translation, so we take W' to be intuitionistic (whereas an important characteristic of W is that its logic is classical).

2 Definition of $\mathtt{T}^{\mathtt{eq}\downarrow}$

The language $\mathtt{T}^{\mathtt{eq}\downarrow}$ is a simple language with natural numbers and dependently-typed recursive functions. The syntax of types T and terms t appears in Figure 1. The variable x is bound in t in the term $\lambda\, x \,.\, t$ and in T' in the type $\Pi^{\theta} x : T . T'$. As explained below, θ for Π-types represents the latent effect of the function's computation (it does not describe the input argument). The variables f and x are bound in t in the term $\mathbf{rec}\, f(x) = t$. We use the notation $[\, t' \,/\, x \,]\, T$ and $[\, t' \,/\, x \,]\, t$ to denote the capture-avoiding substitution of t' for x in types and terms respectively.

We deliberately omit from $\mathtt{T}^{\mathtt{eq}\downarrow}$ many important type-theoretic features which we believe to be orthogonal to the central ideas explored here. A full-fledged type theory based on these ideas would include user-defined inductive types, type polymorphism, perhaps a universe hierarchy, large eliminations, implicit products, and so forth. Some of these features, in particular large eliminations, raise serious technical challenges for this approach (and many others). For this paper we develop the core ideas needed for distinguishing total and possibly partial computations with our effect system and using termination casts to internalize termination, leaving other problems to future work.

2.1 Operational semantics

Reduction for $\mathtt{T}^{\mathtt{eq}\downarrow}$ is defined as a call-by-value small-step operational semantics. Figure 1 presents the syntax of values and evaluation contexts and Figure 2 contains the two judgments that make up this semantics. Values in $\mathtt{T}^{\mathtt{eq}\downarrow}$ include variables, natural numbers, functions and primitive proof terms for the internalized judgments of equality and termination.

We define the reduction rules with two relations: the primitive β rules, written $t \leadsto_{\beta} t'$ describe reduction when a value is in an active position. This relation is used by the main reduction relation $t \leadsto t'$, which lifts beta reduction through evaluation contexts $\mathcal{C}$ and terminates computation for $\mathbf{abort}$, representing finite failure. Other proof forms, including $\mathbf{contra}$, are considered values. We cannot, in fact, obtain a contradiction in the empty context (assuming our theory W' is consistent), but at this point in the development that cannot be shown.

$$\boxed{t \leadsto_\beta t'}$$

$$\frac{}{(\,\lambda\, x\,.\,t\,)\,v \leadsto_\beta [\,v\,/\,x\,]\,t}\ \textsc{Beta\_AppAbs}$$

$$\frac{}{\mathbf{case}\ 0\ t\ t' \leadsto_\beta t}\ \textsc{Beta\_CaseZero}$$

$$\frac{}{\mathbf{case}\ (\,\mathbf{Suc}\,v\,)\ t\ t' \leadsto_\beta t'\ v}\ \textsc{Beta\_CaseSuc}$$

$$\frac{}{(\,\mathbf{rec}\, f(x) = t\,)\,v \leadsto_\beta [\,v\,/\,x\,][\,\mathbf{rec}\, f(x) = t\,/\,f\,]\,t}\ \textsc{Beta\_AppRec}$$

$$\boxed{t \leadsto t'}$$

$$\frac{t \leadsto_\beta t'}{\mathcal{C}\,[\,t\,] \leadsto \mathcal{C}\,[\,t'\,]}\ \textsc{Red\_Ctxt}$$

$$\frac{}{\mathcal{C}\,[\,\mathbf{abort}\,] \leadsto \mathbf{abort}}\ \textsc{Red\_Abort}$$

Figure 2: Call-by-value small-step operational semantics

2.2 Type assignment

Figure 3 defines the *type-assignment* system. The judgment $\Gamma \vdash t : T\ \theta$ states that the term t can be assigned type T in the context Γ with effect θ. (The other two judgments, $\Gamma \vdash \mathbf{Ok}$ and $\Gamma \vdash T$, are used by this one to check that contexts and types are well formed.) We define the system such that θ is an approximation of the termination behavior of the system. If we can derive a judgment $\Gamma \vdash t : T\ \downarrow$, then this means that for any assignment of values to the variables in Γ, reduction of t must terminate. (If the context is inconsistent, t might not terminate even if the type system judges it to do so, since an inconsistent context can make unsatisfiable assertions about termination, which may pollute the type system's judgments.) In contrast, the judgment $\Gamma \vdash t : T\ ?$ places no restrictions on the termination behavior of t. We view θ is as a *capability* on termination behavior [9]. A term with capability $?$ is allowed to diverge, but terms with capability $\downarrow$ cannot. As a result, any term that typechecks with $\downarrow$ will also typecheck with $?$. Thus $?$ is more permissive than $\downarrow$, and we order them as $\downarrow \leq ?$.

Such reasoning is reflected in the type system. $\mathtt{T}^{\mathtt{eq}\downarrow}$ has a call-by-value operational semantics, so variables stand for values. Therefore, a variable is known to terminate, so we can type variables with any effect in rule T_VAR. This pattern occurs often; all terms that are known to terminate have unconstrained effects in the conclusion of their typing rules. In this way, we build subeffecting into the type system and do not need an additional rule to coerce total terms to general ones. Because of this subeffecting, when a premise of a rule uses the general effect, such as K_EQ, it places no restriction on the term.

As is standard in type-and-effect systems, function types are annotated with a *latent effect*. This effect records the termination effect for the body of the function, in rule T_ABS. Likewise, in an application (rule T_APP), the latent effect of the function must be equal or less than the current termination effect. Note that, although the system supports subeffecting, it does not support subtyping. In an application, the type of the argument must exactly match that expected by the function. Although there is a natural extension of subeffecting to subtyping, for simplicity we have not included it in this system.

$\mathtt{T}^{\mathtt{eq}\downarrow}$ types include two propositions. The type $t = t'$ states that two terms are equal and the type $\mathbf{Terminates}\ t$ declares that term t is terminating. The introduction form for the equality proposition (rule T_JOIN) requires both terms to be well typed and evaluate to a common reduct. For flexibility, these terms need not be judged terminating nor have the same type. The elimination form (T_CONV) uses a total proof of equality to convert between equivalent types. Likewise, the introduction form for the $\mathbf{Terminates}\ t$ proposition (T_REIFY) requires showing that the term terminates. Analogously, the elimination form (T_REFLECT) uses a total proof of termination to change the effect of t. $\mathtt{T}^{\mathtt{eq}\downarrow}$ also internalizes an admissible property of the judgment with the empty context—if a term terminates, then

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

$$\frac{\Gamma \vdash \mathbf{Ok}}{\Gamma \vdash \mathbf{nat}}\ \textsc{k\_Nat} \qquad \frac{\Gamma, x : T \vdash T'}{\Gamma \vdash \Pi^{\theta} x : T . T'}\ \textsc{k\_Pi}$$

$$\frac{\Gamma \vdash t : T\ ? \quad \Gamma \vdash t' : T'\ ?}{\Gamma \vdash t = t'}\ \textsc{k\_Eq} \qquad \frac{\Gamma \vdash t : T\ ?}{\Gamma \vdash \mathbf{Terminates}\ t}\ \textsc{k\_Term}$$

$$\boxed{\Gamma \vdash \mathbf{Ok}}$$

$$\frac{}{\cdot \vdash \mathbf{Ok}}\ \textsc{Ok\_empty} \qquad \frac{\Gamma \vdash \mathbf{Ok} \quad \Gamma \vdash T}{\Gamma, x : T \vdash \mathbf{Ok}}\ \textsc{Ok\_cons}$$

$$\boxed{\Gamma \vdash t : T\ \theta}$$

$$\frac{\begin{array}{l} t \leadsto^{*} t_0 \quad t' \leadsto^{*} t_0 \\ \Gamma \vdash t : T\ ? \quad \Gamma \vdash t' : T'\ ? \end{array}}{\Gamma \vdash \mathbf{join} : t = t'\ \theta}\ \textsc{T\_Join} \qquad \frac{\begin{array}{l} \Gamma \vdash t : [\, t_2 \,/\, x \,]\ T\ \theta \\ \Gamma \vdash t' : t_1 = t_2 \downarrow \quad \Gamma \vdash [\, t_1 \,/\, x \,]\ T \end{array}}{\Gamma \vdash t : [\, t_1 \,/\, x \,]\ T\ \theta}\ \textsc{T\_Conv}$$

$$\frac{\Gamma \vdash t : T \downarrow}{\Gamma \vdash \mathbf{terminates} : \mathbf{Terminates}\ t\ \theta}\ \textsc{T\_Reify} \qquad \frac{\begin{array}{l} \Gamma \vdash t : T\ ? \\ \Gamma \vdash t' : \mathbf{Terminates}\ t \downarrow \end{array}}{\Gamma \vdash t : T\ \theta}\ \textsc{T\_Reflect}$$

$$\frac{\Gamma \vdash t : \mathbf{Terminates}\ \mathcal{C}\,[\, t' \,]\ \theta}{\Gamma \vdash t : \mathbf{Terminates}\ t'\ \theta}\ \textsc{T\_CtxTerm} \qquad \frac{\Gamma(x) = T \quad \Gamma \vdash \mathbf{Ok}}{\Gamma \vdash x : T\ \theta}\ \textsc{T\_Var}$$

$$\frac{\Gamma, x : T' \vdash t : T\ \rho \quad \Gamma \vdash \Pi^{\rho} x : T' . T}{\Gamma \vdash \lambda\, x \,.\, t : \Pi^{\rho} x : T' . T\ \theta}\ \textsc{T\_Abs} \qquad \frac{\Gamma \vdash t : \Pi^{\rho} x : T' . T\ \theta \quad \Gamma \vdash t' : T'\ \theta \quad \rho \leq \theta}{\Gamma \vdash t\, t' : [\, t' \,/\, x \,]\ T\ \theta}\ \textsc{T\_App}$$

$$\frac{\Gamma \vdash \mathbf{Ok}}{\Gamma \vdash 0 : \mathbf{nat}\ \theta}\ \textsc{T\_Zero} \qquad \frac{\Gamma \vdash t : \mathbf{nat}\ \theta}{\Gamma \vdash \mathbf{Suc}\, t : \mathbf{nat}\ \theta}\ \textsc{T\_Suc}$$

$$\frac{\Gamma \vdash t : 0 = \mathbf{Suc}\, t' \downarrow}{\Gamma \vdash \mathbf{contra} : T\ \theta}\ \textsc{T\_Contra} \qquad \frac{\Gamma \vdash \mathbf{Ok}}{\Gamma \vdash \mathbf{abort} : T\ ?}\ \textsc{T\_Abort}$$

$$\frac{\Gamma, f : \Pi^{?} x : T' . T , x : T' \vdash t : T\ ?}{\Gamma \vdash \mathbf{rec}\, f(x) = t : \Pi^{?} x : T' . T\ \theta}\ \textsc{T\_Rec} \qquad \frac{\begin{array}{l} \Gamma \vdash t : \mathbf{nat}\ \theta \quad \Gamma \vdash t' : [\, 0 \,/\, x \,]\ T\ \theta \\ \Gamma \vdash t'' : \Pi^{\rho} x' : \mathbf{nat} . [\, \mathbf{Suc}\, x' \,/\, x \,]\ T\ \theta \quad \rho \leq \theta \end{array}}{\Gamma \vdash \mathbf{case}\ t\ t'\ t'' : [\, t \,/\, x \,]\ T\ \theta}\ \textsc{T\_CaseNat}$$

$$\frac{\begin{array}{l} p \notin \mathbf{fv}\, t \\ \Gamma, f : \Pi^{?} x : \mathbf{nat} . T , x : \mathbf{nat} , p : \Pi^{\downarrow} x_1 : \mathbf{nat} . \Pi^{\downarrow} p' : x = \mathbf{Suc}\, x_1 . \mathbf{Terminates}\ (f\ x_1) \vdash t : T \downarrow \end{array}}{\Gamma \vdash \mathbf{rec}\, f(x) = t : \Pi^{\downarrow} x : \mathbf{nat} . T\ \theta}\ \textsc{T\_RecNat}$$

Figure 3: Type assignment system

$$\begin{array}{llll}
\mathit{annot.\ types} & S & ::= & \mathbf{nat} \mid \Pi^{\theta}x\!:\!S.S' \mid a = a' \mid \mathbf{Terminates}\ a \\
\mathit{annot.\ terms} & a & ::= & x \mid a\,a' \mid \lambda^{\theta}x\!:\!S.a \mid 0 \mid \mathbf{Suc}\,a \\
 & & \mid & \mathbf{rec_{nat}}\ f(x\ p)\!: S = a \mid \mathbf{rec}\ f(x\!:\!S)\!: S' = a \mid \mathbf{case}\ x.S\ a\ a'\ a'' \\
 & & \mid & \mathbf{join}\ a\ a' \mid \mathbf{conv}\ x.S\ a'\ a \mid \mathbf{terminates}\ a \mid \mathbf{reflect}\ a\ a' \\
 & & \mid & \mathbf{inv}\ a\ a' \mid \mathbf{contra}\ S\ a \mid \mathbf{abort}\ S
\end{array}$$

Figure 4: Syntax of annotated $\mathtt{T}^{\mathtt{eq}\downarrow}$

the subterm in the active position of the term terminates (T_CTXTERM). This property does not (appear to) follow constructively from the others.

Recursive functions can be typed with either general or total latent effects. In the latter case, the T_RECNAT rule introduces a new hypothesis into the context that may be used to show that the body of the function is total. The assumption $p : \Pi^{\downarrow}x_1 : \mathbf{nat}.\Pi^{\downarrow}p' : x = \mathbf{Suc}\,x_1.\mathbf{Terminates}\ (f\ x_1\ )$ is an assertion that for any number x_1 that is one less than x, the recursive call $(f\ x_1\ )$ terminates. Even though the type of f has a ? latent effect, recursive calls on the immediate predecessor can be cast to be total using this assumption.

The rule T_RECNAT includes a restriction that $p \notin \mathbf{fv}\ t$. This means that the only places that p can occur in a typing derivation is in the proof-premises of T_CONV, T_REFLECT, and T_CONTRA. The advantage of setting up the system this way is that we can define the operational semantics without any reference to proofs: the rule BETA_APPREC does not have to specify a proof term to substitute for free occurrences of p in t. In other words the T_RECNAT rule bakes in a form of *proof erasure* [12, 3, 11].

We may worry that this restriction limits the expressiveness of the language because the variable p can not be used in every context. However, that is not the case as our system satisfies a form of *proof irrelevance*. No matter what proof we have of termination, we can always use the rules T_REIFY and T_REFLECT to replace it by the (computationally) uninformative proof **terminates**. We give an example of this behavior in the next section. Thus, we do not lose anything by making the proof variable p second-class, since we can always replace it with a proof that does not mention p. (Likewise, equality proofs are irrelevant, as we can use T_JOIN followed by T_CONV to show that $\Gamma \vdash u : t = t' \downarrow$ implies $\Gamma \vdash \mathbf{join} : t = t' \downarrow$.)

2.3 Annotated language

The previous two subsections provide a complete specification of the $\mathtt{T}^{\mathtt{eq}\downarrow}$ language. However, in $\mathtt{T}^{\mathtt{eq}\downarrow}$, type inference is not algorithmic. Given a context Γ, a term t and effect θ, it is not clear how to determine if there is some T such that $\Gamma \vdash t : T\ \theta$ holds. The terms do not contain enough information to indicate how to construct a typing derivation.

Fortunately, it is straightforward to produce an annotated version of $\mathtt{T}^{\mathtt{eq}\downarrow}$ where the type checking algorithm is fully determined. Below we give the syntax of the annotated terms. The full typing rules for the annotated system appear in Figure 6. The judgment form is $\Gamma \Vdash a : S\ \theta$, where algorithmically, Γ, a, and θ are inputs to the type checker and type S is the output.

Most annotated term forms have direct correspondence to the unannotated terms. Figure 5 defines the operation $|\cdot|$ that erases annotations. Notably, there are two different forms of recursion, based on which typing rule should be used. Furthermore, the syntax includes terms ($\mathbf{conv}\ x.S\ a'\ a$, $\mathbf{inv}\ a\ a'$, and $\mathbf{reflect}\ a\ a'$) that mark where type conversions, termination inversions and termination casts should occur—these are implicit in the unannotated system.

The annotated system uses types S that are exactly like types T except that they contain annotated

$$
\begin{array}{lcl}
\textit{Types} & & \\
|\,\mathbf{nat}\,| & = & \mathbf{nat} \\
|\,\Pi^{\theta}x\!:\!S.S'\,| & = & \Pi^{\theta}x\!:\!|\,S\,|.|\,S'\,| \\
|\,a\;=\;a'\,| & = & |\,a\,|\;=\;|\,a'\,| \\
|\,\mathbf{Terminates}\;\;a\,| & = & \mathbf{Terminates}\;\;|\,a\,|
\end{array}
$$

$$
\begin{array}{lcllcl}
\textit{Terms} & & & & & \\
|\,x\,| & = & x & |\,\mathbf{join}\;a\;a'\,| & = & \mathbf{join} \\
|\,a\;a'\,| & = & |\,a\,|\,|\,a'\,| & |\,\mathbf{terminates}\;a\,| & = & \mathbf{terminates} \\
|\,\lambda^{\theta}x\!:\!S.a\,| & = & \lambda\,x\,.\,|\,a\,| & |\,\mathbf{contra}\;S\;\;a\,| & = & \mathbf{contra} \\
|\,0\,| & = & 0 & |\,\mathbf{abort}\;S\,| & = & \mathbf{abort} \\
|\,\mathbf{Suc}\;a\,| & = & \mathbf{Suc}\,|\,a\,| & |\,\mathbf{conv}\;x.S\;\;a\;\;a'\,| & = & |\,a\,| \\
|\,\mathbf{case}\;x.S\;\;a\;\;a'\;\;a''\,| & = & \mathbf{case}\,|\,a\,|\,|\,a'\,|\,|\,a''\,| & |\,\mathbf{reflect}\;\;a\;\;a'\,| & = & |\,a\,| \\
|\,\mathbf{rec}_{\mathbf{nat}}\;f(x\;p)\!:\;S=a\,| & = & \mathbf{rec}\,f(x)=|\,a\,| & |\,\mathbf{inv}\;\;a\;\;a'\,| & = & |\,a\,| \\
|\,\mathbf{rec}\;f(x\!:\!S)\!:\;S'=a\,| & = & \mathbf{rec}\,f(x)=|\,a\,| & & &
\end{array}
$$

Figure 5: Annotation erasure

terms. However, because there is no operational semantics defined for annotated terms, the join rule (shown below) first erases the annotations before determining if there is some common reduct. Likewise, the inversion rule uses erasure to find the evaluation context.

Simple comparison of the typing rules of the two systems in a straightforward inductive proof shows that the annotated system is sound and complete with respect to the implicit system.

Proposition 1 (Soundness of annotated system)**.** *If* $\Gamma \Vdash a : S\ \theta$ *then* $\Gamma \vdash |\,a\,| : |\,S\,|\ \theta$.

Proposition 2 (Completeness of annotated system)**.** *If* $\Gamma \vdash t : T\ \theta$ *then there exists an* a *and* S*, such that* $|\,a\,| = t$ *and* $|\,S\,| = T$ *and* $\Gamma \Vdash a : S\ \theta$.

Note that although type inference is syntax-directed, it is only decidable in the annotated system if there is some cut-off in normalization in the join rule. Even if we were to require a and a' to have the total effect in this rule, this restriction would not ensure decidability. An inconsistent context could type a looping term with a total effect. It would be reasonable to make the cutoff part of the annotated $\mathbf{join}$-term itself, although here we use a global cut-off. Note that imposing a cutoff in the join rule in the annotated system does not jeopardize completeness as a single join in the implicit system can be translated to several joins in the annotated system.

Finally, we are not considering the problem of annotation inference for this system. This is an important problem to ease the burden of programming with termination casts. We conjecture that in many simple cases like structural decrease of a single parameter to the function, the appropriate termination casts can be added completely automatically. But working this process out is beyond the scope of this paper.

3 Examples

Natural number addition: internal verification Our first example shows how simple structurally recursive functions can be shown terminating at their definition time using the T_RECNAT rule. We define natural number addition with the following term, showing first its implicit then annotated versions:

$$\boxed{\Gamma \Vdash S}$$

$$\frac{\Gamma \Vdash \mathbf{Ok}}{\Gamma \Vdash \mathbf{nat}}\ \textsc{S\_Nat} \qquad \frac{\Gamma \Vdash S \quad \Gamma\,,\, x\ :\ S \Vdash S'}{\Gamma \Vdash \Pi^{\theta} x\!:\!S.S'}\ \textsc{S\_Pi}$$

$$\frac{\begin{array}{l}\Gamma \Vdash a : S\ ? \quad \Gamma \Vdash a' : S'\ ?\\ \Gamma \Vdash S \quad \Gamma \Vdash S'\end{array}}{\Gamma \Vdash a\ =\ a'}\ \textsc{S\_Eq} \qquad \frac{\Gamma \Vdash a : S\ ?}{\Gamma \Vdash \mathbf{Terminates}\ \ a}\ \textsc{S\_Term}$$

$$\boxed{\Gamma \Vdash \mathbf{Ok}}$$

$$\frac{}{\cdot \Vdash \mathbf{Ok}}\ \textsc{Oka\_empty} \qquad \frac{\Gamma \Vdash \mathbf{Ok} \quad \Gamma \Vdash S}{\Gamma\,,\, x\ :\ S \Vdash \mathbf{Ok}}\ \textsc{Oka\_cons}$$

$$\boxed{\Gamma \Vdash a : S\ \theta}$$

$$\frac{\begin{array}{l}|\,a\,| \leadsto^{N} t \quad |\,a'\,| \leadsto^{N} t\\ \Gamma \Vdash a : S\ ? \quad \Gamma \Vdash a' : S'\ ?\end{array}}{\Gamma \Vdash \mathbf{join}\ a\ a' : a\ =\ a'\ \theta}\ \textsc{AT\_Join} \qquad \frac{\begin{array}{l}\Gamma \Vdash a : [\,a_2\,/\,x\,]\,S\ \theta\\ \Gamma \Vdash a' : a_1\ =\ a_2\ \downarrow \quad \Gamma \Vdash [\,a_1\,/\,x\,]\,S\end{array}}{\Gamma \Vdash \mathbf{conv}\ x.S\ a\ a' : [\,a_1\,/\,x\,]\,S\ \theta}\ \textsc{AT\_Conv}$$

$$\frac{\Gamma \Vdash a : S\ \downarrow}{\Gamma \Vdash \mathbf{terminates}\ a : \mathbf{Terminates}\ \ a\ \theta}\ \textsc{AT\_Reify} \qquad \frac{\Gamma \Vdash a : S\ ? \quad \Gamma \Vdash a' : \mathbf{Terminates}\ \ a\ \downarrow}{\Gamma \Vdash \mathbf{reflect}\ a\ a' : S\ \theta}\ \textsc{AT\_Reflect}$$

$$\frac{\begin{array}{l}\Gamma \Vdash a : \mathbf{Terminates}\ \ a''\ \theta\\ |\,a''\,|\ =\ \mathcal{C}\,[\,|\,a'\,|\,]\end{array}}{\Gamma \Vdash \mathbf{inv}\ a\ a' : \mathbf{Terminates}\ \ a'\ \theta}\ \textsc{AT\_CtxTerm} \qquad \frac{\Gamma(x) = \mathbf{T} \quad \Gamma \Vdash \mathbf{Ok}}{\Gamma \Vdash x : S\ \theta}\ \textsc{AT\_Var}$$

$$\frac{\Gamma\,,\, x\ :\ S' \Vdash a : S\ \rho \quad \Gamma \Vdash \Pi^{\rho} x\!:\!S'.S}{\Gamma \Vdash \lambda^{\rho} x\!:\!S'.a : \Pi^{\rho} x\!:\!S'.S\ \theta}\ \textsc{AT\_Abs} \qquad \frac{\Gamma \Vdash a : \Pi^{\rho} x\!:\!S'.S\ \theta \quad \Gamma \Vdash a' : S'\ \theta \quad \rho\ \leq\ \theta}{\Gamma \Vdash a\ a' : [\,a'\,/\,x\,]\,S\ \theta}\ \textsc{AT\_App}$$

$$\frac{\Gamma \Vdash \mathbf{Ok}}{\Gamma \Vdash 0 : \mathbf{nat}\ \theta}\ \textsc{AT\_Zero} \qquad \frac{\Gamma \Vdash a : \mathbf{nat}\ \theta}{\Gamma \Vdash \mathbf{Suc}\, a : \mathbf{nat}\ \theta}\ \textsc{AT\_Suc}$$

$$\frac{\Gamma \Vdash a : 0\ =\ \mathbf{Suc}\, a'\ \downarrow}{\Gamma \Vdash \mathbf{contra}\ S\ a : S\ \theta}\ \textsc{AT\_Contra} \qquad \frac{\Gamma \Vdash \mathbf{Ok}}{\Gamma \Vdash \mathbf{abort}\ S : S\ ?}\ \textsc{AT\_Abort}$$

$$\frac{\Gamma\,,\, f\ :\ \Pi^{?} x\!:\!S'.S\,,\, x\ :\ S' \Vdash a : S\ ?}{\Gamma \Vdash \mathbf{rec}\ f(x\!:\!S')\!:\ S = a : \Pi^{?} x\!:\!S'.S\ \theta}\ \textsc{AT\_Rec} \qquad \frac{\begin{array}{l}\Gamma \Vdash a : \mathbf{nat}\ \theta \quad \Gamma \Vdash a' : [\,0\,/\,x\,]\,S\ \theta\\ \Gamma \Vdash a'' : \Pi^{\rho} x'\!:\!\mathbf{nat}.[\,\mathbf{Suc}\, x'\,/\,x\,]\,S\ \theta\\ \rho\ \leq\ \theta\end{array}}{\Gamma \Vdash \mathbf{case}\ x.S\ a\ a'\ a'' : [\,a\,/\,x\,]\,S\ \theta}\ \textsc{AT\_CaseNat}$$

$$\frac{\begin{array}{l}p\ \notin\ \mathbf{fv}\ a\\ \Gamma\,,\, f\ :\ \Pi^{?} x\!:\!\mathbf{nat}.S\,,\, x\ :\ \mathbf{nat}\,,\, p\ :\ \Pi^{\downarrow} x_1\!:\!\mathbf{nat}.\Pi^{\downarrow} p'\!:\!x\ =\ \mathbf{Suc}\, x_1.\mathbf{Terminates}\ \ (\,f\, x_1\,) \Vdash a : S\ \downarrow\end{array}}{\Gamma \Vdash \mathbf{rec_{nat}}\ f(x\ p)\!:\ S = a : \Pi^{\downarrow} x\!:\!\mathbf{nat}.S\ \theta}\ \textsc{AT\_RecNat}$$

Figure 6: Annotated type checking system

$$\begin{array}{ll}
\textit{implicit plus} & \overset{\text{def}}{=} \lambda\, x_2 \,.\, \mathbf{rec}\, f(x_1) = (\, \mathbf{case}\; x_1 \; (\, \lambda\, q \,.\, x_2\,)\; (\, \lambda\, x' \,.\, \lambda\, q \,.\, \mathbf{Suc}\,(\, f\; x'\,)\,)\,)\, \mathbf{join} \\
\textit{annotated plus} & \overset{\text{def}}{=} \lambda^{\downarrow} x_2 \!:\! \mathbf{nat}.\; \mathbf{rec}_{\mathbf{nat}}\; f\; (x_1\; p) \!:\, \mathbf{nat} = \\
& \quad (\mathbf{case}\; x.(\, \Pi^{\downarrow} q \!:\! x_1 \; = \; x.\mathbf{nat}\,)\; x_1 \\
& \quad\quad (\, \lambda^{\downarrow} q \!.\, x_1 \; = \; 0.x_2\,) \\
& \quad\quad (\lambda^{\downarrow} x' \!:\! \mathbf{nat}.\lambda^{\downarrow} q \!.\, x_1 \; = \; \mathbf{Suc}\, x'.\; \mathbf{Suc}\,(\, \mathbf{reflect}\; (\, f\; x'\,)\; (\, p\, x'\, q\,)\,))) \\
& \quad (\, \mathbf{join}\; x_1\; x_1\,)
\end{array}$$

In this example, we must abstract over equality types that are then applied to **join**. This standard trick, used frequently in Coq and similar dependent type theories, introduces different assumptions of equalities into the context, depending on the case branch. As remarked above, we have deliberately omitted from $\mathtt{T}^{\mathtt{eq}\downarrow}$ a number of features that would improve some of these examples, notably implicit products (as proposed by Miquel [11]) for equality proofs in case-terms.

The typing rules verify that plus is a total operation. For example, in the annotated system we can show:

$$\cdot \Vdash \textit{plus} : \Pi^{\downarrow} x_1 \!:\! \mathbf{nat}.\Pi^{\downarrow} x_2 \!:\! \mathbf{nat}.\mathbf{nat}\; \downarrow$$

To see why this is so, consider the context that we use to type check the body of the recursive function:

$$\Gamma \overset{\text{def}}{=} x_1 \;:\; \mathbf{nat}\,,\; x_2 \;:\; \mathbf{nat}\,,\; f \;:\; \Pi^{?} x_1 \!:\! \mathbf{nat}.\mathbf{nat}\,,\; p \;:\; \Pi^{\downarrow} x' \!:\! \mathbf{nat}.\Pi^{\downarrow} q \!:\! x_1 \; = \; \mathbf{Suc}\, x'.\mathbf{Terminates}\; (\, f\; x'\,)\,,\; \cdot$$

In this context, we would like to show that the case expression has type $(\, \Pi^{\downarrow} q \!:\! x_1 \; = \; x_1.\mathbf{nat}\,)$. Note that the abstraction of q must be $\downarrow$ so that when we apply the case expression to **join** the entire expression will have the $\downarrow$ effect. In the zero case, we use rules TA_ABS and TA_VAR to show that the abstraction has the desired total function type.

In the successor case, we use a termination cast to show that the recursive call is total. Without this cast, we would be unable to use the latent effect $\downarrow$ in the abstraction of q. Using the rules for variables and application we can show that the recursive call has a general effect, but by itself, this will not let us define a total function.

$$\Gamma\,,\; x' \;:\; \mathbf{nat}\,,\; q \;:\; x_1 \; = \; \mathbf{Suc}\, x' \Vdash f\; x' : \mathbf{nat}\; ?$$

However, given the extra argument from recursive function, we can produce a proof that the recursive call terminates.

$$\Gamma\,,\; x' \;:\; \mathbf{nat}\,,\; q \;:\; x_1 \; = \; \mathbf{Suc}\, x' \Vdash p\, x'\, q : \mathbf{Terminates}\; (\, f\; x'\,)\; \downarrow$$

From these two, we can use a termination cast to change the effect of the recursive call.

$$\Gamma\,,\; x' \;:\; \mathbf{nat}\,,\; q \;:\; x_1 \; = \; \mathbf{Suc}\, x' \Vdash \mathbf{reflect}\; (\, f\; x'\,)\; (\, p\, x'\, q\,) : \mathbf{nat}\; \downarrow$$

Finally, we can use the rules for successor and abstraction to conclude that the successor case has the desired type.

Natural number addition: external verification An advantage of this system is that we do not need to prove that plus is total when we define it. We could also define plus using general recursion:

$$\textit{plus} \overset{\text{def}}{=} \lambda\, x_2 \,.\, \mathbf{rec}\, f(x_1) = \mathbf{case}\; x_1\; x_2\; (\, \lambda\, z \,.\, \mathbf{Suc}\,(\, f\; z\,)\,)$$

But note, the best typing derivation will assign a ? latent effect to this function. (For brevity, this and further examples will be presented in the implicit language.)

$$\cdot \vdash \mathit{plus} : \Pi^{\downarrow} x_2 : \mathbf{nat} . \Pi^{?} x_1 : \mathbf{nat} . \mathbf{nat} \ \downarrow$$

However, all is not lost. We can still prove the following theorem and use it in a termination cast to show that a particular application of *plus* terminates. The proof term (below) uses recursion to construct a total witness for this theorem.

$$\begin{array}{lcl} \mathit{plustotal} & : & \Pi^{\downarrow} x_2 : \mathbf{nat} . \Pi^{\downarrow} x_1 : \mathbf{nat} . \mathbf{Terminates}\ (\, \mathit{plus}\, x_2\, x_1\,) \\ \mathit{plustotal} & \overset{\text{def}}{=} & \lambda\, x_2 \,.\, (\, \mathbf{rec}\, f(x_1) = (\, \mathbf{case}\ x_1\ (\, \lambda\, q \,.\, \mathbf{terminates}\,)\ (\, \lambda\, z \,.\, \lambda\, q \,.\, \mathbf{terminates}\,)\,)\, \mathbf{join}\,) \end{array}$$

To understand this proof term, we look at the typing derivation in each branch of the case term. Let Γ be the context that rule T_RECNAT uses to check the body of the recursive definition, shown below.

$$\begin{array}{rrcl} \Gamma \overset{\text{def}}{=} & x_2 & : & \mathbf{nat}, \\ & x_1 & : & \mathbf{nat}, \\ & f & : & \Pi^{?} z : \mathbf{nat} . \mathbf{Terminates}\ (\, \mathit{plus}\, x_2\, z\,), \\ & p & : & \Pi^{\downarrow} z : \mathbf{nat} . \Pi^{\downarrow} q : x_1 \ = \ \mathbf{Suc}\, z . \mathbf{Terminates}\ (\, f\ z\,) \end{array}$$

Then in the zero case, because $\mathit{plus}\, x_2\, 0$ evaluates to x_2 and variables terminate, we can use rule T_CONV to show that case total.

$$\dfrac{\dfrac{\dfrac{\Gamma\, ,\ q\ :\ x_1\ =\ 0 \vdash x_2 : \mathbf{nat}\ \downarrow}{\Gamma\, ,\ q\ :\ x_1\ =\ 0 \vdash \mathbf{terminates} : \mathbf{Terminates}\ \ x_2\ \downarrow} \qquad \dfrac{\vdots}{\Gamma \vdash \mathbf{join} : \mathit{plus}\, x_2\, 0\ =\ x_2\ \downarrow}}{\Gamma\, ,\ q\ :\ x_1\ =\ 0 \vdash \mathbf{terminates} : \mathbf{Terminates}\ \ (\, \mathit{plus}\, x_2\, 0\,)\ \downarrow}}{\Gamma \vdash \lambda\, q \,.\, \mathbf{terminates} : \Pi^{\downarrow} q : x_1\ =\ 0 . \mathbf{Terminates}\ \ (\, \mathit{plus}\, x_2\, 0\,)\ \downarrow}$$

For the successor case, we need to make a recursive call to the theorem to show that the recursive call to the function terminates. Below, let Γ' be the extended environment $\Gamma\, ,\ z\ :\ \mathbf{nat}\, ,\ q\ :\ x_1\ =\ \mathbf{Suc}\, z$ and $(*)$ be the derivation of $\Gamma' \vdash \mathbf{join} : \mathit{plus}\, x_2\, (\, \mathbf{Suc}\, z\,)\ =\ \mathbf{Suc}\, (\, \mathit{plus}\, x_2\, z\,)\ \downarrow$. Then, the derivation looks like:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma' \vdash \mathit{plus}\, x_2\, z : \mathbf{nat}\ ?} \qquad \dfrac{\vdots}{\Gamma' \vdash f\ z : \mathbf{Terminates}\ \ (\, \mathit{plus}\, x_2\, z\,)\ \downarrow}}{\Gamma' \vdash \mathit{plus}\, x_2\, z : \mathbf{nat}\ \downarrow}}{\Gamma' \vdash \mathbf{Suc}\, (\, \mathit{plus}\, x_2\, z\,) : \mathbf{nat}\ \downarrow}}{\Gamma' \vdash \mathbf{terminates} : \mathbf{Terminates}\ \ (\, \mathbf{Suc}\, (\, \mathit{plus}\, x_2\, z\,)\,)\ \downarrow} \qquad (*)}{\Gamma' \vdash \mathbf{terminates} : \mathbf{Terminates}\ \ (\, \mathit{plus}\, x_2\, (\, \mathbf{Suc}\, z\,)\,)\ \downarrow}$$

First-class termination proofs Recursive functions can also call helper functions in their definitions, passing off the recursive term and a proof that the recursive call will terminate. For example, suppose there is some function h that takes a an argument, a (general) function to call on that argument, and a proof that the call terminates.

$$h : \Pi^{\downarrow} x : \mathbf{nat} . \Pi^{\downarrow} f : \Pi^{?} x : \mathbf{nat} . \mathbf{nat} . \Pi^{\downarrow} p : \mathbf{Terminates}\ \ (\, f\ x\,) . \mathbf{nat}$$

For example, h may just apply f to x and use a termination cast to show the effect total. We can use h in the definition of a total recursive function, even if we do not know its definition. (Let Γ be a context which contains the above binding for h.)

$$\Gamma \vdash \mathbf{rec}\, f(x) = (\,\mathbf{case}\; x\; (\,\lambda\, q\,.\,0\,)\; (\,\lambda\, z\,.\,\lambda\, q\,.\,h\, z\, f\, \mathbf{terminates}\,)\,)\,\mathbf{join} : \Pi^{\downarrow} x\!:\!\mathbf{nat}.\mathbf{nat}\ \downarrow$$

Note that in this example, we use $\mathbf{terminates}$ as the proof that $f\ z$ terminates. Although T_RECNAT introduces the variable p, of type $\Pi^{\downarrow} z : \mathbf{nat}.\Pi^{\downarrow} q : z \;=\; \mathbf{Suc}\, z.\mathbf{Terminates}\ \ (\,f\ z\,)$, we cannot pass $p\ z\ q$ as the termination proof to h because p cannot be mentioned in the term. However, the proof term $\mathbf{terminates}$ works instead, as shown by the following derivation. (Let Γ' be the context in the successor case, i.e. Γ extended with bindings for x, f, p, z and q.)

$$\cfrac{\cfrac{\cfrac{\vdots}{\Gamma' \vdash p\ z\ q : \mathbf{Terminates}\ \ (\,f\ z\,)\ \downarrow} \qquad \cfrac{\vdots}{\Gamma' \vdash f\ z : \mathbf{nat}\ ?}}{\Gamma' \vdash f\ z : \mathbf{nat}\ \downarrow}\ \text{T\_REFLECT}}{\Gamma' \vdash \mathbf{terminates} : \mathbf{Terminates}\ \ (\,f\ z\,)\ \downarrow}\ \text{T\_REIFY}$$

Natural number division Finally, we demonstrate a function that requires a course-of-values argument to show termination: natural number division. The general problem is that division calls itself recursively on a number that is smaller, but is not the direct predecessor of the argument. To show that this function terminates, we do structural recursion on an upper bound of the dividend instead of the dividend itself. (Note that we could also define division as a possibly partial function, without this extra upper-bound argument, and separately write a proof that states that division is a total function.) The type we use for division is:

$$\mathit{div} : \Pi^{\downarrow} z\!:\!\mathbf{nat}.\Pi^{\downarrow} x\!:\!\mathbf{nat}.\Pi^{\downarrow} x'\!:\!\mathbf{nat}.\Pi^{\downarrow} u\!:\!(\,\mathit{lte}\, x'\, x\,) \;=\; \mathbf{true}.\mathbf{nat}$$

where z is the divisor, x' is the dividend, x is an upper bound of the dividend, and lte is a function that determines if the first number is "less-than-or-equal" the second. We have been parsimonious in omitting a boolean type, so we use 0 and $\mathbf{Suc}\, 0$ for $\mathbf{false}$ and $\mathbf{true}$, respectively in the result of lte. Therefore, we define

$$\mathit{lte} \overset{\text{def}}{=} \mathbf{rec}\, f(x) = \lambda\, u\,.\,\mathbf{case}\; x\; (\,\mathbf{Suc}\, 0\,)\; (\,\lambda\, x'\,.\,\mathbf{case}\; u\; 0\; (\,f\; x'\,)\,)$$

and show

$$\cdot \vdash \mathit{lte} : \Pi^{?} x\!:\!\mathbf{nat}.\Pi^{?} x'\!:\!\mathbf{nat}.\mathbf{nat}\ \downarrow$$

Note that we are considering lte as a possibly partial function; nothing is harmed by not requiring it to be total. We also define cut-off subtraction as a total function minus of type $\Pi^{\downarrow} x\!:\!\mathbf{nat}.\Pi^{\downarrow} x'\!:\!\mathbf{nat}.\mathbf{nat}$ (details omitted). The code for division is then:

$$\begin{array}{ll} \mathit{div} \overset{\text{def}}{=} & \lambda z.((\mathbf{case}\; z \\ & \quad (\,\lambda\, q\,.\,\lambda\, x\,.\,\lambda\, x'\,.\,\lambda\, u\,.\,0\,) \\ & \quad (\,\lambda\, z'\,.\,\lambda\, q\,.\,\mathbf{rec}\, f(x) = \lambda\, x'\,.\,\lambda\, u\,.\,(\,(\,\mathbf{case}\;(\,\mathit{lte}\,(\,\mathbf{Suc}\, x\,)\; z\,)\; t_1\; (\,\lambda\, z''\,.\,\lambda\, q'\,.\,0\,)\,)\,\mathbf{join}\,)\,)) \\ & \mathbf{join}) \end{array}$$

We handle the case of division by 0 up front, obtaining an assumption $q \;:\; z \;=\; \mathbf{Suc}\, z'$ when the divisor is not zero. Next, we case split on whether or not the bound x is strictly less than z; that is, $\mathit{lte}\,(\,\mathbf{Suc}\, x\,)\, z$. If so, we use the term $\lambda\, z''\,.\,\lambda\, q'\,.\,0$ of type

$$\Pi^{\downarrow} z''\!:\!\mathbf{nat}.\Pi^{\downarrow} q'\!:\!\mathit{lte}\,(\,\mathbf{Suc}\, x\,)\, z \;=\; (\,\mathbf{Suc}\, z''\,).\mathbf{nat}$$

Then the quotient is 0. If not, we use the term t_1, of type $\Pi^{\downarrow} q' : (\, \mathit{lte}\,(\, \mathbf{Suc}\, x\,)\, z \;=\; 0\,).\mathbf{nat}$, which is (with t_2 discussed below):

$$t_1 \;\overset{\mathrm{def}}{=}\; \lambda\, q' \,.\, (\, \mathbf{Suc}\,(\, f\,(\, \mathit{pred}\, x\,)\,(\, \mathit{minus}\, x'\, z\,)\, t_2\,)\,)$$

In this case, we are decreasing our bound on the dividend by one, and then using a termination cast to show that $f\,(\, \mathit{pred}\, x\,)$ is terminating. Here, we define pred as just $\lambda\, x \,.\, \mathbf{case}\; x\; 0\; \lambda\, x' \,.\, x'$. Of course, since this is the implicit language, the termination cast does not appear in the term itself. To apply the termination cast, we must use the implicit assumption p telling us that f terminates on the predecessor of x. We can prove that $\mathbf{case}\; x\; 0\; \lambda\, x' \,.\, x'$ is the predecessor of x in this case, because the assumptions $q : z \;=\; (\, \mathbf{Suc}\, z'\,)$ and $q' : \mathit{lte}\,(\, \mathbf{Suc}\, x\,)\, z \;=\; \mathbf{false}$ show that x is non-zero: Intuitively, q' implies that x is greater than or equal to z, which we know is non-zero by q. The term t_2 is a proof that $\mathit{minus}\, x'\, z$ is less than or equal to the predecessor of the bound, $\mathbf{case}\; x\; 0\; \lambda\, x' \,.\, x'$. In fact, $\mathbf{join}$ will serve for t_2 because the desired equation is provable from the assumptions.

4 A Logical Semantics for $\mathtt{T}^{\mathtt{eq}\downarrow}$

In this section, we give a semantics for $\mathtt{T}^{\mathtt{eq}\downarrow}$ in terms of a simple constructive logic called W'. This semantics informs our design of $\mathtt{T}^{\mathtt{eq}\downarrow}$ and can potentially be used as part of a consistency proof for $\mathtt{T}^{\mathtt{eq}\downarrow}$. The theory W' is reminiscent of Feferman's theory W (see, for example, Chapter 13 of [10]). W is a classical second-order theory of general-recursive functions, classified by class terms which correspond to simple types. W supports quantification over class terms, and quantification over defined individual terms. It is defined in Beeson's Logic of Partial Terms, a logic designed for reasoning about definedness in the presence of partial functions [5]. W includes a relatively weak form of natural-number induction. Indeed, W is conservative over Peano Arithmetic.

4.1 The theory W'

Figure 7 gives the syntax for sorts A (which are just simple types) and formulas F for the theory W'; as well as typing contexts Σ and contexts H for logical assumptions. Terms t are just as for (implicit) $\mathtt{T}^{\mathtt{eq}\downarrow}$, except without $\mathbf{contra}$, $\mathbf{terminates}$, and $\mathbf{join}$. Figure 8 gives the proof rules for the theory W'. The form of judgments is $\Sigma\,;\; H \;\vdash\; F$. This expresses that formula F holds under the assumed formulas in H. Σ is a typing context declaring free term-level variables occurring in H and F.

W' is similar in spirit to Feferman's W, but differs in a number of details. First, W is a two-sorted theory: there is a sort for individual terms, and one for class terms. To express that term t is in class C, theory W uses an atomic formula $t \in C$. Our theory W', in contrast, is a multi-sorted first-order logic, with one sort for every simple type. So W' does not make use of a predicate symbol to express that a term has a sort. We only insist that terms are well-sorted when instantiating quantifiers. This is apparent in the rule PV_ALLE, which depends on a simple typing judgment for W'. The rules for this typing judgment may be found in the companion technical report [18]. Well-formedness of equations does not require well-sortedness of the terms in W' (as also in W). Also, we have no reason at the moment to include non-constructive reasoning in W', so we define it using principles of intuitionistic logic only.

A few more words on the proof principles of W' are warranted. The PV_OPSEM equates terms t and t' iff $t \rightsquigarrow^{*} t'$. Thanks to the PV_SUBST rule, symmetry and transitivity of equality can be derived in a standard way. We do not require quantifiers to be instantiated by only terminating terms. This means that for induction principles, we must state explicitly that the terms in question are terminating. We include a principle PV_COMPIND of computational induction, on the structure of a terminating computation. That is, if we know that an application of a recursive function is terminating, we can prove

$$\begin{array}{lcl}
A & ::= & \mathbf{nat} \mid A \to A' \\
F & ::= & \mathbf{True} \mid \forall x : A.F \mid F \Rightarrow F' \mid F \wedge F' \mid \mathbf{Terminates}\ t \mid t = t' \\
\Sigma & ::= & \cdot \mid \Sigma\,, x : A \\
H & ::= & \cdot \mid H\,, F
\end{array}$$

Figure 7: Simple types, formulas, typing contexts, and assumption contexts of W'

$$\frac{F \in H}{\Sigma\,; H \vdash F}\ \textsc{Pv\_Assume} \qquad \frac{\Sigma\,, x : A\,; H \vdash F \quad x \notin \mathbf{fv}\,H}{\Sigma\,; H \vdash \forall x : A.F}\ \textsc{Pv\_AllI}$$

$$\frac{\Sigma\,; H \vdash \forall x : A.F \quad \Sigma \vdash t : A}{\Sigma\,; H \vdash [\,t\,/\,x\,]\,F}\ \textsc{Pv\_AllE} \qquad \frac{\Sigma\,; H\,, F \vdash F'}{\Sigma\,; H \vdash F \Rightarrow F'}\ \textsc{Pv\_ImpI}$$

$$\frac{\Sigma\,; H \vdash F \Rightarrow F' \quad \Sigma\,; H \vdash F}{\Sigma\,; H \vdash F'}\ \textsc{Pv\_ImpE} \qquad \frac{\Sigma\,; H \vdash F \quad \Sigma\,; H \vdash F'}{\Sigma\,; H \vdash F \wedge F'}\ \textsc{Pv\_AndI}$$

$$\frac{\Sigma\,; H \vdash F \wedge F'}{\Sigma\,; H \vdash F}\ \textsc{Pv\_AndE1} \qquad \frac{\Sigma\,; H \vdash F \wedge F'}{\Sigma\,; H \vdash F'}\ \textsc{Pv\_AndE2}$$

$$\frac{}{\Sigma\,; H \vdash \mathbf{True}}\ \textsc{Pv\_TrueI} \qquad \frac{\Sigma\,; H \vdash 0 = \mathbf{Suc}\,t}{\Sigma\,; H \vdash F}\ \textsc{Pv\_Contra}$$

$$\frac{t \rightsquigarrow^{*} t'}{\Sigma\,; H \vdash t = t'}\ \textsc{Pv\_OpSem} \qquad \frac{\Sigma\,; H \vdash t = t' \quad \Sigma\,; H \vdash [\,t\,/\,x\,]\,F}{\Sigma\,; H \vdash [\,t'\,/\,x\,]\,F}\ \textsc{Pv\_Subst}$$

$$\frac{}{\Sigma\,; H \vdash \mathbf{Terminates}\ 0}\ \textsc{Pv\_Term0} \qquad \frac{\Sigma\,; H \vdash \mathbf{Terminates}\ t}{\Sigma\,; H \vdash \mathbf{Terminates}\ \mathbf{Suc}\,t}\ \textsc{Pv\_TermS}$$

$$\frac{}{\Sigma\,; H \vdash \mathbf{Terminates}\ \lambda\,x\,.\,t}\ \textsc{Pv\_TermAbs} \qquad \frac{}{\Sigma\,; H \vdash \mathbf{Terminates}\ \mathbf{rec}\,f(x) = t}\ \textsc{Pv\_TermRec}$$

$$\frac{\Sigma\,; H \vdash \mathbf{Terminates}\ \mathcal{C}\,[\,t\,]}{\Sigma\,; H \vdash \mathbf{Terminates}\ t}\ \textsc{Pv\_TermInv} \qquad \frac{\Sigma\,; H \vdash \mathbf{Terminates\ abort}}{\Sigma\,; H \vdash F}\ \textsc{Pv\_NotTermAbort}$$

$$\frac{\Sigma\,; H \vdash [\,0\,/\,x\,]\,F \quad \Sigma\,, x' : \mathbf{nat}\,; H\,, \mathbf{Terminates}\ x'\,, [\,x'\,/\,x\,]\,F \vdash [\,\mathbf{Suc}\,x'\,/\,x\,]\,F}{\Sigma\,; H \vdash \forall x : \mathbf{nat}.\mathbf{Terminates}\ x \Rightarrow F}\ \textsc{Pv\_Ind}$$

$$\frac{\Sigma\,, f : A' \to A\,; H\,, \forall x : A'.[\,f\,x\,/\,z\,]\,F \vdash \forall x : A'.[\,t\,/\,z\,]\,F \quad \Sigma \vdash \mathbf{rec}\,f(x) = t : A' \to A}{\Sigma\,; H \vdash \forall x : A'.\mathbf{Terminates}\ (\,\mathbf{rec}\,f(x) = t\,)\,x \Rightarrow [\,(\,\mathbf{rec}\,f(x) = t\,)\,x\,/\,z\,]\,F}\ \textsc{Pv\_CompInd}$$

Figure 8: Theory W'

a property of such an application by assuming it is true for recursive calls, and showing it is true for an outer arbitrary call of the function. Note that the assumption of termination of the application of the recursive function is essential: without it, we could prove diverging terms terminate. We also include a principle Pv_TermInv of computational inversion, which allows us to conclude $\mathbf{Terminates}\ t$ from $\mathbf{Terminates}\ \mathcal{C}\,[\,t\,]$. Interestingly, even without the inversion rule of $\mathtt{T}^{\mathrm{eq}\downarrow}$, the theorem we prove below would make heavy use of computational inversion. In a classical theory like W, this principle may well be derivable from the other axioms. Here, it does not seem to be.

$$
\begin{array}{lcl@{\qquad}lcl}
[\![x]\!]^C & = & x & [\![t\,t']\!]^C & = & [\![t]\!]^C\ [\![t']\!]^C \\
[\![\lambda\,x\,.\,t]\!]^C & = & \lambda x.\,[\![t]\!]^C & [\![0]\!]^C & = & 0 \\
[\![\mathbf{Suc}\,t]\!]^C & = & \mathbf{S}\ [\![t]\!]^C & [\![\mathbf{join}]\!]^C & = & 0 \\
[\![\mathbf{terminates}]\!]^C & = & 0 & [\![\mathbf{contra}]\!]^C & = & 0 \\
[\![\mathbf{abort}]\!]^C & = & \mathbf{abort} & [\![\mathbf{rec}\,f(x) = t]\!]^C & = & \mathbf{rec}\,f(x).[\![t]\!]^C \\
[\![\mathbf{case}\ t\ t'\ t'']\!]^C & = & \mathbf{C}\ [\![t]\!]^C\ [\![t']\!]^C\ [\![t'']\!]^C & & &
\end{array}
$$

Figure 9: Computational translation of terms

$$
\begin{array}{rcl@{\qquad}rcl}
[\![\mathbf{nat}]\!]^C & = & \mathbf{nat} & [\![\mathbf{nat}]\!]^L\ t & = & \mathbf{True} \\
[\![\Pi^\theta x : T.\,T']\!]^C & = & [\![T]\!] \to [\![T']\!] & [\![\Pi^\theta x : T.\,T']\!]^L\ t & = & \forall x : [\![T]\!]^C.[\![T]\!]^L_\downarrow\ x\ \Rightarrow [\![T']\!]^L_\theta\ (t\ x) \\
[\![t\ =\ t']\!]^C & = & \mathbf{nat} & [\![t_1\ =\ t_2]\!]^L\ t & = & [\![t_1]\!]^C = [\![t_2]\!]^C \\
[\![\mathbf{Terminates}\ \ t]\!]^C & = & \mathbf{nat} & [\![\mathbf{Terminates}\ \ t']\!]^L\ t & = & \mathbf{Terminates}\ [\![t']\!]^C
\end{array}
$$

$$
\begin{array}{rcl}
[\![T]\!]^L_\downarrow\ t & = & \mathbf{Terminates}\ t\ \wedge\ [\![T]\!]^L\ t \\
[\![T]\!]^L_?\ t & = & \mathbf{Terminates}\ t\ \Rightarrow\ [\![T]\!]^L\ t
\end{array}
$$

Figure 10: Interpretation of types

Computational translation of terms Figure 9 defines what we will refer to as the computational translation of $\mathtt{T}^{\mathtt{eq}\downarrow}$ terms (the “C” is for computational). This translation, which is almost trivial, just maps logical terms $\mathbf{join}$, $\mathbf{terminates}$, and $\mathbf{contra}$ to 0.

Translation of types Next, given $\mathtt{T}^{\mathtt{eq}\downarrow}$ type T, we define $[\![T]\!]^C$ and $[\![T]\!]^L$. The “L” is for logical translation. This $[\![T]\!]^C$ is a sort A, and $[\![T]\!]^L$ is a predicate on translated terms. Recall that the syntax for such types and for the formulas F used in such predicates is defined in Figure 7 above. The definition of the interpretations is then given in Figure 10. Note that one can confirm the well-foundedness of this definition by expanding the definition of $[\![\,T]\!]^L_\theta$, a convenient abbreviation, wherever it is used.

4.2 Examples

Example 1. If we consider the type $\Pi^\downarrow x_1 : \mathbf{nat}.\Pi^\downarrow x_2 : \mathbf{nat}.\mathbf{nat}$, we will get the following. Note that the assumptions below that variables terminate reflect the call-by-value nature of the language. A translation for a call-by-name language would presumably not include such assumptions.

$$
\begin{array}{rcl}
[\![\Pi^\downarrow x_1 : \mathbf{nat}.\Pi^\downarrow x_2 : \mathbf{nat}.\mathbf{nat}]\!]^C & = & \mathbf{nat} \to (\,\mathbf{nat} \to \mathbf{nat}\,) \\
[\![\Pi^\downarrow x_1 : \mathbf{nat}.\Pi^\downarrow x_2 : \mathbf{nat}.\mathbf{nat}]\!]^L\ \mathit{plus} & = & \forall x_1 : \mathbf{nat}.\ \mathbf{Terminates}\ x_1\ \wedge \mathbf{True} \Rightarrow \\
& & \qquad \mathbf{Terminates}\ (\mathit{plus}\ x_1) \wedge \\
& & \qquad \forall x_2 : \mathbf{nat}.\ \mathbf{Terminates}\ x_2 \wedge \mathbf{True} \Rightarrow \\
& & \qquad\qquad \mathbf{Terminates}\ (\mathit{plus}\ x_1\ x_2) \wedge\ \mathbf{True}
\end{array}
$$

Example 2 (higher-order, total). If we wanted to type a function *iter* which iterates a terminating function x_1, starting from x_2, and does this iteration x_3 times, we might use the type: $\Pi^\downarrow x_1 : \Pi^\downarrow x : \mathbf{nat}.\mathbf{nat}.\Pi^\downarrow x_2 : \mathbf{nat}.\Pi^\downarrow x_3 : \mathbf{nat}.\mathbf{nat}$. For this type (call it T for brevity), we will get the following

$$\begin{array}{lcllcl} [\![\cdot]\!]^C & = & \cdot & \qquad [\![\cdot]\!]^L & = & \cdot \\ [\![\Gamma\, , \, x \, : \, T]\!]^C & = & [\![\Gamma]\!], x : [\![T]\!]^C & \qquad [\![\Gamma\, , \, x \, : \, T]\!]^L & = & [\![\Gamma]\!], \, [\![T]\!]^L_{\downarrow} \, x \end{array}$$

Figure 11: Interpretation of contexts

translations:

$$\begin{array}{rcl} [\![T]\!]^C & = & (\, \mathbf{nat} \to \mathbf{nat}\,) \to (\, \mathbf{nat} \to (\, \mathbf{nat} \to \mathbf{nat}\,)\,) \\ [\![T]\!]^L \; \textit{iter} & = & \forall x_1 : \mathbf{nat} \to \mathbf{nat}.\, \mathbf{Terminates}\; x_1 \wedge \\ & & \qquad (\forall x : \mathbf{nat}. \mathbf{Terminates}\;\; x \;\wedge\; \mathbf{True} \Rightarrow \mathbf{Terminates}\; (x_1\; x) \;\wedge\; \mathbf{True}) \;\Rightarrow \\ & & \qquad\qquad\qquad \mathbf{Terminates}\; (\textit{iter}\; x_1) \;\wedge \\ & & \forall x_2 : \mathbf{nat}.\, \mathbf{Terminates}\; x_2 \wedge \mathbf{True} \Rightarrow \mathbf{Terminates}\; (\textit{iter}\; x_1\; x_2) \;\wedge \\ & & \forall x_3 : \mathbf{nat}.\, \mathbf{Terminates}\; x_3 \wedge \mathbf{True} \Rightarrow \mathbf{Terminates}\; (\textit{iter}\; x_1\; x_2\; x_3) \;\wedge\; \mathbf{True} \end{array}$$

Notice that in this case, the logical interpretation $[\![T]\!]^L$ includes a hypothesis that the function x_1 is terminating. This corresponds to the fact that x_1 has type $\Pi^{\downarrow} x : \mathbf{nat}.\mathbf{nat}$ in the original $\mathtt{T}^{\mathtt{eq}\downarrow}$ type.

Example 3 (higher-order, partial). If we wanted to type a different version of *iter* which, when given a general-recursive function x_1 and a starting value x_2, returns a general-recursive function taking input x_3 and iterating x_1 x_3 times starting from x_2, we might use the type: $\Pi^{\downarrow} x_1 : \Pi^{?} x : \mathbf{nat}.\mathbf{nat}.\Pi^{\downarrow} x_2 : \mathbf{nat}.\Pi^{?} x_3 : \mathbf{nat}.\mathbf{nat}$. For this type (call it T), we will get the following logical translation:

$$\begin{array}{rcl} [\![T]\!]^L \; \textit{iter} & = & \forall x_1 : \mathbf{nat} \to \mathbf{nat}.\, \mathbf{Terminates}\; x_1 \wedge \\ & & \qquad (\forall x : \mathbf{nat}. \mathbf{Terminates}\;\; x \;\wedge\; \mathbf{True} \Rightarrow \mathbf{Terminates}\; (x_1\; x) \;\Rightarrow\; \mathbf{True}) \;\Rightarrow \\ & & \qquad\qquad\qquad \mathbf{Terminates}\; (\textit{iter}\; x_1) \;\wedge \\ & & \forall x_2 : \mathbf{nat}.\, \mathbf{Terminates}\; x_2 \wedge \mathbf{True} \Rightarrow \mathbf{Terminates}\; (\textit{iter}\; x_1\; x_2) \;\wedge \\ & & \forall x_3 : \mathbf{nat}.\, \mathbf{Terminates}\; x_3 \wedge \mathbf{True} \Rightarrow \mathbf{Terminates}\; (\textit{iter}\; x_1\; x_2\; x_3) \;\Rightarrow\; \mathbf{True} \end{array}$$

4.3 Translation of contexts

Figure 11 gives a similar 2-part translation of typing contexts. The translation $[\![\cdot]\!]^C$ produces a simple-typing context Σ, while the translation $[\![\cdot]\!]^L$ produces a logical context H, which asserts, for each variable x, that x terminates and has the property given by the $[\![\cdot]\!]^L$ translation of its type.

4.4 Translation of typing judgments

We are now in a position to state the main theorems of this paper. The proofs are given in the companion technical report. Theorem 4 shows that the logical translation of types is sound: the property expressed by $[\![T]\!]^L_\theta$ can indeed be proved to hold for the translation $[\![t]\!]^C$ of terms of type T.

Theorem 3 (Soundness of Computational Translation)**.** *If* $\Gamma \vdash t \,:\, T\; \theta$*, then* $[\![\Gamma]\!]^C \vdash [\![t]\!]^C : [\![T]\!]^C$.

Theorem 4 (Soundness of Logical Translation)**.** *If* $\Gamma \vdash t \,:\, T\; \theta$*, then* $[\![\Gamma]\!]^C; [\![\Gamma]\!]^L \vdash [\![T]\!]^L_\theta\; [\![t]\!]^C$.

5 Related Work

Capretta's Partiality Monad Capretta [7] gives an account of general recursion in terms of a coinductive type constructor $(\cdot)^\nu$, and many $\mathtt{T}^{\mathtt{eq}\downarrow}$ programs can be fairly mechanically translated into programs using $(\cdot)^\nu$ by a translation similar to the the one described by Wadler and Thiemann [19]. However, one interesting difference is that $\mathtt{T}^{\mathtt{eq}\downarrow}$ functions can have a return type which depends on a potentially nonterminating argument. It is not clear how to represent this in a monadic framework.

For example, if we imagine a version of $\mathtt{T}^{\mathtt{eq}\downarrow}$ extended with option types, and suppose we are given a decision procedure for equality of $\mathbf{nat}$s and a partial function which computes the minimum zero of a function:

$$\begin{array}{l} \mathit{eqDec} : \Pi^{\downarrow} x : \mathbf{nat}.\Pi^{\downarrow} x' : \mathbf{nat}.\mathbf{Maybe}\,(\,x \,=\, x'\,) \\ \mathit{minZero} : \Pi^{?} f : (\,\Pi^{\downarrow} x : \mathbf{nat}.\mathbf{nat}\,).\mathbf{nat} \end{array}$$

Then we can easily compose these to make a function to test if two functions have the same least zero:

$$\begin{array}{l} \lambda f\,.\,\lambda f'\,.\,\mathit{eqDec}\,(\,\mathit{minZero}\,f\,)\,(\,\mathit{minZero}\,f'\,) \\ \quad : \Pi^{\downarrow} f : (\,\Pi^{\downarrow} x : \mathbf{nat}.\mathbf{nat}\,).\Pi^{?} f' : (\,\Pi^{\downarrow} x : \mathbf{nat}.\mathbf{nat}\,).\mathbf{Maybe}\,(\,\mathit{minZero}\,f \,=\, \mathit{minZero}\,f'\,) \end{array}$$

However the naive translation of this into monadic form,

$$\lambda f.\lambda f'.(\mathit{minZero}\ f) \gg\!= (\lambda m.(\mathit{minZero}\ f') \gg\!= (\lambda m'.\mathbf{return}\ (\mathit{eqDec}\ m\ m'))),$$

is not well typed, since the monadic bind $\gg\!= \,: \forall A\ B.\, A^\nu \to (A \to B^\nu) \to B^\nu$ does not have a way to propagate the type dependency.

Other Another approach, not depending on coinductive types, is explored by Capretta and Bove, who define a special-purpose accessibility predicate for each general-recursive function, and then define the function by structural recursion on the proof of accessibility for the function's input [6]. ATS and GURU both separate the domains of proofs and programs, and can thus allow general recursion without endangering logical soundness [17, 8]. Systems like Cayenne [2], ΩMEGA [15]. and CONCOQTION [13] support dependent types and general recursion, but do not seek to identify a fragment of the term language which is sound as a proof system (although CONCOQTION uses COQ proofs for reasoning about type indices).

6 Conclusion

$\mathtt{T}^{\mathtt{eq}\downarrow}$ combines equality types and general recursion, using an effect system to distinguish total from possibly partial terms. Termination casts are used to change the type system's view of the termination behavior of a term. Like other casts, termination casts have no computational relevance and are erased in passing from the annotated to the implicit type system. We have given a logical semantics for $\mathtt{T}^{\mathtt{eq}\downarrow}$ in terms of a multi-sorted first-order theory of general-recursive functions. Future work includes further meta-theory, including type soundness for $\mathtt{T}^{\mathtt{eq}\downarrow}$ and further analysis of the proposed theory W'; as well as incorporation of other typing features, in particular polymorphism and large eliminations. An important further challenge is devising algorithms to reconstruct annotations in simple cases or for common programming idioms.

Acknowledgments. Many thanks to the PAR 2010 reviewers for an exceptionally close reading and many constructive criticisms. All syntax definitions in this paper were typeset and type-checked with the OTT tool [14]. Thanks also to other members of the TRELLYS project, especially Tim Sheard, for helpful conversations on these ideas. This work was partially supported by the U.S. National Science Foundation under grants 0702545, 0910510 and 0910786.

References

[1] Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.

[2] Lennart Augustsson. Cayenne–a language with dependent types. In *Proc. 3rd ACM International Conference on Functional Programming (ICFP)*, pages 239–250, 1998.

[3] B. Barras and B. Bernardo. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008*, volume 4962 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2008.

[4] G. Barthe, M. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

[5] M. Beeson. *Foundations of Constructive Mathematics: Metamathematical Studies*. Springer, 1985.

[6] A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15:671–708, February 2005. Cambridge University Press.

[7] V. Capretta. General Recursion via Coinductive Types. *Logical Methods in Computer Science*, 1(2):1–28, 2005.

[8] C. Chen and H. Xi. Combining Programming with Theorem Proving. In *Proceedings of the 10th International Conference on Functional Programming (ICFP05)*, Tallinn, Estonia, September 2005.

[9] K. Crary, D. Walker, and G. Morrisett. Typed Memory Management in a Calculus of Capabilities. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275. ACM, 1999.

[10] S. Feferman. *In the Light of Logic*. Oxford University Press, 1998.

[11] A. Miquel. The Implicit Calculus of Constructions. In *Typed Lambda Calculi and Applications*, pages 344–359, 2001.

[12] N. Mishra-Linger and T. Sheard. Erasure and Polymorphism in Pure Type Systems. In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference (FOSSACS)*, pages 350–364. Springer, 2008.

[13] E Pasalic, J. Siek, W. Taha, and S. Fogarty. Concoqtion: Indexed Types Now! In G. Ramalingam and E. Visser, editors, *ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation*, 2007.

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

[15] T. Sheard. Type-Level Computation Using Narrowing in Ωmega. In *Programming Languages meets Program Verification*, 2006.

[16] M. Sozeau. Subset Coercions in Coq. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, pages 237–252, 2006.

[17] A. Stump, M. Deters, A. Petcher, T. Schiller, and T. Simpson. Verified Programming in Guru. In T. Altenkirch and T. Millstein, editors, *Programming Languges meets Program Verification (PLPV)*, 2009.

[18] Aaron Stump, Vilhelm Sjöberg, and Stephanie Weirich. Termination casts: A flexible approach to termination with general recursion (technical appendix). Technical Report MS-CIS-10-21, Computer and Information Science, University of Pennsylvania, May 2010.

[19] P. Wadler and P. Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, 2003.