1-1-2010

Termination Casts: A Flexible Approach to Termination With General Recursion (Technical Appendix)

Aaron Stump  
*University of Iowa*

Vilhelm Sjoberg  
*University of Pennsylvania*

Stephanie Weirich  
*University of Pennsylvania, sweirich@cis.upenn.edu*

Follow this and additional works at: [http://repository.upenn.edu/cis_reports](http://repository.upenn.edu/cis_reports)

Recommended Citation


This paper is posted at ScholarlyCommons. [http://repository.upenn.edu/cis_reports/930](http://repository.upenn.edu/cis_reports/930)

For more information, please contact libraryrepository@pobox.upenn.edu.
Abstract
This paper proposes a type-and-effect system called $\mathcal{T}_{\text{eq}} \downarrow$, 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 $\mathcal{T}_{\text{eq}} \downarrow$ to a logical theory of termination for general recursive, simply typed functions. Every typing judgment of $\mathcal{T}_{\text{eq}} \downarrow$ is translated to a theorem expressing the appropriate termination property of the computational part of the $\mathcal{T}_{\text{eq}} \downarrow$ term.

Comments

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/930
Abstract

This paper proposes a type-and-effect system called $\text{T}_{\text{eq}}$, 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 $\text{T}_{\text{eq}}$ to a logical theory of termination for general recursive, simply typed functions. Every typing judgment of $\text{T}_{\text{eq}}$ is translated to a theorem expressing the appropriate termination property of the computational part of the $\text{T}_{\text{eq}}$ 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 $(\cdot)^
u$ 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 $\mathsf{T^{eql}}$ 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 $\Gamma \vdash t : T\downarrow$ includes a termination effect $\theta$, which can be either $\downarrow$ (called “total”), for terms that are known to terminate, or $\uparrow$ (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 [18]. 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, $\mathsf{T^{eql}}$ provides a flexible approach to termination because the judgment of totality, $\Gamma \vdash t : T\downarrow$, is internalized into the type system. The type $\mathsf{Terminates} t$ 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 $\mathsf{Terminates} t$. 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 $\mathsf{T^{eql}}$. Because type assignment is not algorithmic for $\mathsf{T^{eql}}$, we also develop an annotated version of $\mathsf{T^{eql}}$ 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
Termination Casts

Section 3. Next, in Section 4 we develop our central meta-theoretic result, based on a translation of \( \text{T_{eq}^\downarrow} \) typing judgments to judgments about termination of the term in question, formulated in 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 \( \text{T_{eq}^\downarrow} \) 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 \( \text{T_{eq}^\downarrow} \)

The language \( \text{T_{eq}^\downarrow} \) 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, \( \theta \) for \( \Pi \)-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 \( \text{rec} f(x) = t \). 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 \( \text{T_{eq}^\downarrow} \) 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 \( \text{T_{eq}^\downarrow} \) 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 \( \text{T_{eq}^\downarrow} \) 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 \( \beta \) rules, written \( t \sim_\beta t' \) describe reduction when a value is in an active position. This relation is used by the main reduction relation \( t \sim t' \), which lifts beta reduction through evaluation contexts \( C \) and terminates computation for \( \text{abort} \), representing finite failure. Other proof forms, including \( \text{contra} \), 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.
### 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 $\Gamma$ with effect $\theta$. (The other two judgments, $\Gamma \vdash \text{Ok}$ and $\Gamma \vdash T$, are used by this one to check that contexts and types are well formed.) We define the system such that $\theta$ 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 $\Gamma$, 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 $\Gamma \vdash t : T \uparrow$ places no restrictions on the termination behavior of $t$. We view $\theta$ as a *capability* on termination behavior [9]. A term with capability $\uparrow$ is allowed to diverge, but terms with capability $\downarrow$ cannot. As a result, any term that typechecks with $\downarrow$ will also typecheck with $\uparrow$. Thus $\uparrow$ is more permissive than $\downarrow$, and we order them as $\downarrow \leq \uparrow$.

Such reasoning is reflected in the type system. $\mathsf{T_{eq\uparrow}}$ 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 $\mathsf{T_{VAR}}$. 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 $\mathsf{K_{EQ}}$, 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 $\mathsf{T_{ABS}}$. Likewise, in an application (rule $\mathsf{T_{APP}}$), 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.

$\mathsf{T_{eq\downarrow}}$ types include two propositions. The type $t = t'$ states that two terms are equal and the type $\mathsf{Terminates} t$ declares that term $t$ is terminating. The introduction form for the equality proposition (rule $\mathsf{T_{JOIN}}$) 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. 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.

$\mathsf{T_{eq\uparrow}}$ types include two propositions. The type $t = t'$ states that two terms are equal and the type $\mathsf{Terminates} t$ declares that term $t$ is terminating. The introduction form for the equality proposition (rule $\mathsf{T_{JOIN}}$) 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 $\mathsf{Terminates} t$ 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 $t$.

$\mathsf{T_{eq\downarrow}}$ also internalizes an admissible property of the judgment with the empty context—if a term terminates, then...
\[
\begin{align*}
& \Gamma \vdash T \\
& \quad \Gamma \vdash \text{Ok} \quad \text{K\_NAT} \\
& \quad \Gamma \vdash \text{nat} \quad \text{K\_NAT} \\
& \quad \Gamma \vdash t : T \quad \Gamma \vdash t' : T' \quad \text{K\_EQ} \\
& \quad \Gamma \vdash t = t' \\
& \quad \Gamma \vdash \text{Terminates} t \\
& \quad \Gamma \vdash \text{Ok} \\
& \quad \Gamma \vdash T \\
& \quad \Gamma, x : T \vdash \text{Ok} \\
& \quad \Gamma \vdash t : T \\
& \quad \Gamma \vdash [t_2/x]T \quad \text{T\_Conv} \\
& \quad \Gamma \vdash t' : [t_1/x]T \\
& \quad \Gamma \vdash t' : [t_1/x]T \quad \text{T\_Reify} \\
& \quad \Gamma \vdash t : T \\
& \quad \Gamma \vdash x : T \quad \text{T\_Var} \\
& \quad \Gamma \vdash t : [t'/x]T \quad \text{T\_App} \\
& \quad \Gamma \vdash t : \text{nat} \quad \text{T\_Succ} \\
& \quad \Gamma \vdash \text{Ok} \\
& \quad \Gamma \vdash 0 : \text{nat} \quad \text{T\_Zero} \\
& \quad \Gamma \vdash t = 0 \quad \text{T\_Contra} \\
& \quad \Gamma \vdash \text{Suc} t' \quad \text{T\_Rec} \\
& \quad \Gamma \vdash \text{rec} f(x) = t : \text{nat} \quad \text{T\_RecNat} \\
& \quad \Gamma \vdash \text{Ok} \\
& \quad \Gamma \vdash \text{abort} : T \\
& \quad \Gamma, x : \text{nat}, p : \Pi^x_1: \text{nat}, \Pi^p_1 : x = \text{Succ}, t : T \quad \text{T\_CaseNat} \\
\end{align*}
\]

Figure 3: Type assignment system
 Recursive functions can be typed with either general or total latent effects. In the latter case, the $\text{T.RECNAT}$ rule introduces a new hypothesis into the context that may be used to show that the body of the function is total. The assumption $p : \Pi^{x_1 : \text{nat}}. \Pi^{p' : x} : \text{Suc} x_1. \text{Terminates} (f x_1)$ 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 $\text{T.RECNAT}$ includes a restriction that $p \notin \text{fv} t$. This means that the only places that $p$ can occur in a typing derivation is in the proof-premises of $\text{T.CONV}$, $\text{T.REFLECT}$, and $\text{T.CONTRA}$. The advantage of setting up the system this way is that we can define the operational semantics without any reference to proofs: the rule $\text{BETA.APPREC}$ does not have to specify a proof term to substitute for free occurrences of $p$ in $t$. In other words the $\text{T.RECNAT}$ 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 $p$ cannot 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 $\text{T.REIFY}$ and $\text{T.REFLECT}$ to replace it by the (computationally) uninformative proof $\text{terminates}$. 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 $\text{T.JOIN}$ followed by $\text{T.CONV}$ to show that $\Gamma \vdash u : t = t' \Downarrow$ implies $\Gamma \vdash \text{join} : t = t' \Downarrow$.)

### 2.3 Annotated language

The previous two subsections provide a complete specification of the $\mathcal{T}^{\text{eql}}$ language. However, in $\mathcal{T}^{\text{eql}}$, type inference is not algorithmic. Given a context $\Gamma$, a term $t$ and effect $\theta$, 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 $\mathcal{T}^{\text{eql}}$ 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, $\Gamma$, $a$, and $\theta$ 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 (conv $x.S a' a$, inv $a a'$, and 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 terms. However, because there is no operational semantics defined for annotated terms, the join rule...
Termination Casts

Types

| nat                   | = nat                |
| Π₀θ x:S.S'           | = Π₀θ x:S|S' |
| a = a'               | = |a| = |a'| |
| Terminates a         | = Terminates |a|

Terms

| x                     | = x                    |
| a a'                  | = |a| |a'|                    |
| λ₀θ x:S.a            | = λ x.|a|                    |
| 0                    | = 0                    |
| Suc a                | = Suc |a|                    |
| case x:S a a' a''    | = case |a| |a'| |a''|                    |
| recnat f(x p): S = a | = rec f(x) = |a|                    |
| rec f(x:S): S' = a   | = rec f(x) = |a|                    |

Figure 5: Annotation erasure

Proposition 1 (Soundness of annotated system). If Γ ⊩ a : S θ then Γ ⊢ |a| : |S| θ.

Proposition 2 (Completeness of annotated system). If Γ ⊢ t : T θ then there exists an a and S, such that |a| = t and |S| = T and Γ ⊩ a : S θ.

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 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_REC_NAT rule. We define natural number addition with the following term, showing first its implicit then annotated versions:

 Term dimensions:

Types

| nat       | = nat                |
| Π₀θ x:S.S' | = Π₀θ x:S|S' |
| a = a'    | = |a| = |a'| |
| Terminates a | = Terminates |a|

Terms

| x       | = x                    |
| a a'    | = |a| |a'|                    |
| λ₀θ x:S.a | = λ x.|a|                    |
| 0       | = 0                    |
| Suc a   | = Suc |a|                    |
| case x:S a a' a'' | = case |a| |a'| |a''|                    |
| recnat f(x p): S = a | = rec f(x) = |a|                    |
| rec f(x:S): S' = a   | = rec f(x) = |a|                    |

Figure 5: Annotation erasure

Proposition 1 (Soundness of annotated system). If Γ ⊩ a : S θ then Γ ⊢ |a| : |S| θ.

Proposition 2 (Completeness of annotated system). If Γ ⊢ t : T θ then there exists an a and S, such that |a| = t and |S| = T and Γ ⊩ a : S θ.

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 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_REC_NAT rule. We define natural number addition with the following term, showing first its implicit then annotated versions:
Figure 6: Annotated type checking system
**implicit plus** \[\def \text{implicit plus} { \lambda x_2. \text{rec } f(x_1) = (\text{case } x_1 (\lambda q.x_2) (\lambda x'. \lambda q. \text{Suc}(f x'))) \text{join} }\]

**annotated plus** \[\def \text{annotated plus} { \lambda \downarrow x_2. \text{nat. rec}_\text{nat } f (x_1 p): \text{nat} = (\text{case } x. (\Pi^1 q:x_1 = x. \text{nat}) x_1 (\lambda \downarrow q.x_1 = 0.x_2) (\lambda \downarrow x': \text{nat. } \lambda \downarrow q.x_1 = \text{Suc}. x'. \text{Suc } (\text{reflect } (f x') (p x' q))))) (\text{join } x_1 x_1) }\]

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 \(T_{=\text{eq}}\) 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:

\[\vdash \text{plus} : \Pi^1 x_1 : \text{nat. } \Pi^1 x_2 : \text{nat. nat} \downarrow\]

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

\[\Gamma \defeq x_1 : \text{nat. } x_2 : \text{nat. } f : \Pi^1 x_1 : \text{nat. nat. } p : \Pi^1 x' : \text{nat. } \Pi^1 q : x_1 = \text{Suc}. x'. \text{Terminates } (f x')\]

In this context, we would like to show that the case expression has type \((\Pi^1 q : x_1 = x_1. \text{nat.})\). 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.

\[\Gamma, x' : \text{nat. } q : x_1 = \text{Suc} x' \vdash f x' : \text{nat.} \]

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

\[\Gamma, x' : \text{nat. } q : x_1 = \text{Suc} x' \vdash p x' q : \text{Terminates } (f x') \downarrow\]

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

\[\Gamma, x' : \text{nat. } q : x_1 = \text{Suc} x' \vdash \text{reflect } (f x') (p x' q) : \text{nat} \downarrow\]

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:

\[\text{def } \text{plus} \defeq \lambda x_2. \text{rec } f(x_1) = \text{case } x_1 x_2 (\lambda z. \text{Suc}(f z))\]

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.)

\[\vdash \text{plus} : \Pi^1 x_2 : \text{nat. } \Pi^2 x_1 : \text{nat. nat} \downarrow\]
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 \( \text{plus} \) terminates. The proof term (below) uses recursion to construct a total witness for this theorem.

\[
\begin{align*}
\text{plustotal} &\quad : \quad \Pi^x_2 : \text{nat}. \Pi^x_1 : \text{nat} \cdot \text{Terminates} \left( \text{plus} x_2 x_1 \right) \\
\text{plustotal} &\quad \triangleq \quad \lambda x_2. \left( \text{rec f} (x_1) = \left( \text{case} x_1 \left( \lambda q. \text{terminates} \right) \left( \lambda z. \lambda q. \text{terminates} \right) \right) \text{join} \right)
\end{align*}
\]

To understand this proof term, we look at the typing derivation in each branch of the case term. Let \( \Gamma \) be the context that rule T.RECNAT uses to check the body of the recursive definition, shown below.

\[
\Gamma \triangleq \begin{array}{l}
x_2 : \text{nat}, \\
x_1 : \text{nat}, \\
f : \Pi^x_2 : \text{nat} \cdot \text{Terminates} \left( \text{plus} x_2 x_2 \right), \\
p : \Pi^x_1 : \text{nat}. \Pi^x_1 q : x_1 = \text{Suc} z \cdot \text{Terminates} \left( f z \right)
\end{array}
\]

Then in the zero case, because \( \text{plus} x_2 0 \) evaluates to \( x_2 \) and variables terminate, we can use rule T.CONV to show that case total.

\[
\begin{array}{l}
\Gamma, q : x_1 = 0 \vdash x_2 : \text{nat} \\
\vdots \\
\Gamma, q : x_1 = 0 \vdash \text{terminates} : \text{Terminates} x_2 \\
\vdots \\
\Gamma \vdash \text{join} : \text{plus} x_2 0 = x_2 \\
\vdots \\
\Gamma \vdash \lambda q. \text{terminates} : \Pi^x_1 q : x_1 = 0 \cdot \text{Terminates} \left( \text{plus} x_2 0 \right)
\end{array}
\]

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 \( \Gamma' \) be the extended environment \( \Gamma, z : \text{nat}, q : x_1 = \text{Suc} z \) and \((*)\) be the derivation of \( \Gamma' \vdash \text{join} : \text{plus} x_2 \left( \text{Suc} z \right) = \text{Suc} \left( \text{plus} x_2 z \right) \). Then, the derivation looks like:

\[
\begin{array}{l}
\vdots \\
\vdots \\
\Gamma' \vdash \text{plus} x_2 z : \text{nat} \\
\Gamma' \vdash f z : \text{Terminates} \left( \text{plus} x_2 z \right) \\
\Gamma' \vdash \text{plus} x_2 z : \text{nat} \\
\Gamma' \vdash \text{Suc} \left( \text{plus} x_2 z \right) : \text{nat} \\
\Gamma' \vdash \text{terminates} : \text{Terminates} \left( \text{Suc} \left( \text{plus} x_2 z \right) \right) \\
\vdots \\
\Gamma' \vdash \text{terminates} : \text{Terminates} \left( \text{plus} x_2 \left( \text{Suc} z \right) \right)
\end{array}
\]

**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.

\[
h : \Pi^x_1 : \text{nat}. \Pi^x_2 : \text{nat} \cdot \Pi^x_1 f : \text{Terminates} \left( f x \right) \cdot \text{nat}
\]

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 \( \Gamma \) be a context which contains the above binding for \( h \).)

\[
\Gamma \vdash \text{rec f} (x) = \left( \text{case} x \left( \lambda q. 0 \right) \left( \lambda z. \lambda q. h z f \text{terminates} \right) \right) \text{join} : \Pi^x_1 : \text{nat} \cdot \text{nat}
\]

Note that in this example, we use \( \text{terminates} \) as the proof that \( f z \) terminates. Although T_RECNAT introduces the variable \( p \), of type \( \Pi^x_1 : \text{nat}. \Pi^x_1 q : z = \text{Suc} z \cdot \text{Terminates} \left( f z \right) \), we cannot pass \( p z q \) as the
termination proof to \( h \) because \( p \) cannot be mentioned in the term. However, the proof term \text{terminates} works instead, as shown by the following derivation. (Let \( \Gamma' \) be the context in the successor case, i.e. \( \Gamma \) extended with bindings for \( x, f, p, z \) and \( q \).)

\[
\begin{align*}
\vdash p z q : \text{Terminates} \ (f z) & \quad \vdash f z : \text{nat} \ ? \\
\Gamma' & \vdash \text{Terminates} \ (f z) \downarrow \\
\Gamma' & \vdash \text{Terminates} \ (f z) \downarrow \\
\Gamma' & \vdash \text{Terminates} \ (f z) \downarrow
\end{align*}
\]

\textbf{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:

\[
div : \Pi^1 z : \text{nat} . \Pi^1 x : \text{nat} . \Pi^1 x' : \text{nat} . \Pi^1 u : (\text{lte} x' x) = \text{true} . \text{nat}
\]

where \( z \) is the divisor, \( x' \) is the dividend, \( x \) is an upper bound of the dividend, and \( \text{lte} \) 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 \text{Suc}0 for \text{false} and \text{true}, respectively in the result of \text{lte}. Therefore, we define

\[
\text{lte} \overset{\text{def}}{=} \text{rec} f (x) = \lambda . u . \text{case} x (\text{Suc}0) (\lambda . x' . \text{case} u 0 (f x'))
\]

and show

\[
\vdash \text{lte} : \Pi^1 x : \text{nat} . \Pi^1 x' : \text{nat} . \text{nat} \downarrow
\]

Note that we are considering \( \text{lte} \) 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 \text{minus} of type \( \Pi^1 x : \text{nat} . \Pi^1 x' : \text{nat} . \text{nat} \) (details omitted). The code for division is then:

\[
div \overset{\text{def}}{=} \lambda z . ( (\text{case} z \\
(\lambda q . \lambda x . \lambda x' . \lambda u . 0) \\
(\lambda z' . \lambda q . \text{rec} f (x) = \lambda x' . \lambda u . ( (\text{case} (\text{lte} (\text{Suc} x) z) t1 (\lambda z'' . \lambda q' . 0)) \text{join}) ))) \\
\text{join})
\]

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

\[
\Pi^1 z'' : \text{nat} . \Pi^1 q' : \text{lte} (\text{Suc} x) z = (\text{Suc} z'') . \text{nat}
\]

Then the quotient is 0. If not, we use the term \( t1 \), of type \( \Pi^1 q' : (\text{lte} (\text{Suc} x) z = 0) . \text{nat} \), which is (with \( t2 \) discussed below):

\[
t1 \overset{\text{def}}{=} \lambda q' . (\text{Suc} (\text{pred} x) (\text{minus} x' z) t2)
\]

In this case, we are decreasing our bound on the dividend by one, and then using a termination cast to show that \( f (\text{pred} x) \) is terminating. Here, we define \( \text{pred} \) as just \( \lambda x . \text{case} x 0 \lambda x'. x' \). 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 case $x \ 0 \ \lambda x'. x'$ is the predecessor of $x$ in this case, because the assumptions $q : z = (\text{Suc} z')$ and $q' : \text{ite} (\text{Suc} x) z = \text{false}$ 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 $\text{minus} x' z$ is less than or equal to the predecessor of the bound, case $x \ 0 \ \lambda x'. x'$. In fact, join will serve for $t_2$ because the desired equation is provable from the assumptions.

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

In this section, we give a semantics for $\text{T}^{\text{eq}}$ in terms of a simple constructive logic called $W'$. This semantics informs our design of $\text{T}^{\text{eq}}$ and can potentially be used as part of a consistency proof for $\text{T}^{\text{eq}}$. 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 $\Sigma$ and contexts $H$ for logical assumptions. Terms $t$ are just as for (implicit) $\text{T}^{\text{eq}}$, except without contra, terminates, and 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 $H$. $\Sigma$ 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 $\text{PV}_{\text{ALLE}}$, which depends on a simple typing judgment for $W'$. The rules for this typing judgment may be found in the appendix (Section C). 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 $\text{PV}_{\text{OPSEM}}$ equates terms $t$ and $t'$ iff $t \sim^* t'$. Thanks to the $\text{PV}_{\text{SUBST}}$ 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 $\text{PV}_{\text{COMPIND}}$ 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 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 $\text{PV}_{\text{TERMINV}}$ of computational inversion, which allows us to conclude terminates $t$ from terminates $\in \{t\}$. Interestingly, even without the inversion rule of $\text{T}^{\text{eq}}$, 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.
$A ::= \text{nat} \mid A \rightarrow A'$

$F ::= \text{True} \mid \forall x : A. F \mid F \Rightarrow F' \mid F \land F' \mid \text{Terminates } t \mid t = t'$

$\Sigma ::= \cdot \mid \Sigma, x : A$

$H ::= \cdot \mid H, F$

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

$F \in H$

$\Sigma; H \vdash F$

$\Sigma, x : A; H \vdash F \quad x \notin \text{fv } H$

$\Sigma; H \vdash \forall x : A. F$

$\Sigma; H \vdash \text{true}$

$t \sim^* t'$

$\Sigma; H \vdash t = t'$

$\Sigma; H \vdash \text{terminates } 0$

$\Sigma; H \vdash \text{terminates } \lambda x.t$

$\Sigma; H \vdash \text{terminates } \mathcal{C}[t]$

$\Sigma; H \vdash \text{terminates } t$

$\Sigma; H \vdash \text{terminates } \text{rec } f(x) = t$

$\Sigma; H \vdash \text{terminates } \text{abort}$

$\Sigma; H \vdash 0 \in \text{nat}$

$\Sigma, x : \text{nat}; H, \text{terminates } x', [x'/x]F \vdash [\text{Suc } x'/x]F$

$\Sigma; H \vdash \forall x : \text{nat. terminates } x \Rightarrow F$

$\Sigma; f : A' \rightarrow A; H, \forall x : A', [f x/z]F \vdash \forall x : A', [t/z]F$

$\Sigma; H \vdash \forall x : A'. \text{terminates } (\text{rec } f(x) = t) x \Rightarrow [(\text{rec } f(x) = t) x/z]F$

$\Sigma; f : A' \rightarrow A; H, \forall x : A', [f x/z]F \vdash \forall x : A', [t/z]F$

$\Sigma; H \vdash \forall x : A'. \text{terminates } (\text{rec } f(x) = t) x \Rightarrow [(\text{rec } f(x) = t) x/z]F$

$\Sigma; f : A' \rightarrow A; H, \forall x : A', [f x/z]F \vdash \forall x : A', [t/z]F$

$\Sigma; H \vdash \forall x : A'. \text{terminates } (\text{rec } f(x) = t) x \Rightarrow [(\text{rec } f(x) = t) x/z]F$

Figure 8: Theory $W'$
Termination Casts

If we consider the type $\Pi^1 x : \text{nat}. \Pi^1 y : \text{nat}. \text{nat}$, 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.

$$\llbracket \Pi^1 x : \text{nat}. \Pi^1 y : \text{nat}. \text{nat} \rrbracket^C = \text{nat} \rightarrow (\text{nat} \rightarrow \text{nat})$$

$$\llbracket \Pi^1 x : \text{nat}. \Pi^1 y : \text{nat}. \text{nat} \rrbracket^L t = \text{Terminates } \llbracket t \rrbracket^L t$$

$$\llbracket \Pi^1 x : \text{nat}. \Pi^1 y : \text{nat}. \text{nat} \rrbracket^L t = \text{Terminates } \llbracket t \rrbracket^C$$

**Example 1.** If we wanted to type a function $\text{iter}$ which iterates a terminating function $x_1$, starting from $x_2$, and does this iteration $x_3$ times, we might use the type: $\Pi^1 x_1 : \Pi^1 x : \text{nat}. \Pi^1 x_2 : \text{nat}$. The "L" is for logical translation. This $\llbracket T \rrbracket^C$ is a sort $A$, and $\llbracket T \rrbracket^L$ 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 above. Note that one can confirm the well-foundedness of this definition by expanding the definition of $\llbracket T \rrbracket^L_\theta$, a convenient abbreviation, wherever it is used.

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

**Translation of types** Next, given $T^e$ type $T$, we define $\llbracket T \rrbracket^C$ and $\llbracket T \rrbracket^L$. The “L” is for logical translation. This $\llbracket T \rrbracket^C$ is a sort $A$, and $\llbracket T \rrbracket^L$ 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 above. Note that one can confirm the well-foundedness of this definition by expanding the definition of $\llbracket T \rrbracket^L_\theta$, a convenient abbreviation, wherever it is used.

**4.2 Examples**

**Example 2 (higher-order, total).** If we wanted to type a function $\text{iter}$ which iterates a terminating function $x_1$, starting from $x_2$, and does this iteration $x_3$ times, we might use the type: $\Pi^1 x_1 : \Pi^1 x : \text{nat}. \Pi^1 x_2 : \text{nat}$. The "L" is for logical translation. This $\llbracket T \rrbracket^C$ is a sort $A$, and $\llbracket T \rrbracket^L$ 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 above. Note that one can confirm the well-foundedness of this definition by expanding the definition of $\llbracket T \rrbracket^L_\theta$, a convenient abbreviation, wherever it is used.

**Example 1.** If we consider the type $\Pi^1 x : \text{nat}. \Pi^1 y : \text{nat}. \text{nat}$, 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.
Theorem 3 shows that the logical translation of types is sound: the property expressed by $\llbracket t \rrbracket^C_\theta$ can indeed be proved to hold for the translation $\llbracket t \rrbracket^C$ of terms of type $T$.

**Theorem 3 (Soundness of Computational Translation).** If $\Gamma \vdash t : T \theta$, then $\llbracket \Gamma \rrbracket^C \vdash \llbracket t \rrbracket^C : \llbracket T \rrbracket^C$.

**Theorem 4 (Soundness of Logical Translation).** If $\Gamma \vdash t : T \theta$, then $\llbracket \Gamma \rrbracket^C_\theta \vdash \llbracket t \rrbracket^C_\theta : \llbracket T \rrbracket^C_\theta$. 

---

**nat.Π^1 x : nat.nat.** For this type (call it $T$ for brevity), we will get the following logical translations:

$$\begin{align*}
\llbracket T \rrbracket^C & = (\text{nat} \rightarrow \text{nat}) \rightarrow (\text{nat} \rightarrow (\text{nat} \rightarrow \text{nat})) \\
\llbracket T \rrbracket^L_{\text{iter}} & = \forall x_1 : \text{nat} \rightarrow \text{nat}. \text{Terminates } x_1 \land \\
& \quad (\forall x : \text{nat}. \text{Terminates } x \land \text{True} \Rightarrow \text{Terminates } (x_1 x) \land \text{True}) \Rightarrow \\
& \quad \text{Terminates } (\text{iter } x_1) \land \\
& \quad \forall x_2 : \text{nat}. \text{Terminates } x_2 \land \text{True} \Rightarrow \text{Terminates } (\text{iter } x_1 x_2) \land \\
& \quad \forall x_3 : \text{nat}. \text{Terminates } x_3 \land \text{True} \Rightarrow \text{Terminates } (\text{iter } x_1 x_2 x_3) \land \text{True}
\end{align*}$$

Notice that in this case, the logical interpretation $\llbracket t \rrbracket^L$ includes a hypothesis that the function $x_1$ is terminating. This corresponds to the fact that $x_1$ has type $\Pi^1 x : \text{nat} \rightarrow \text{nat}$ in the original $T_{\text{eq}}$ type.

**Example 3 (higher-order, partial).** If we wanted to type a different version of $\text{iter}$ 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: $\Pi^1 x_1 : \Pi^1 x_2 : \text{nat} \rightarrow \text{nat}$. For this type (call it $T$), we will get the following logical translation:

$$\begin{align*}
\llbracket T \rrbracket^L_{\text{iter}} & = \forall x_1 : \text{nat} \rightarrow \text{nat}. \text{Terminates } x_1 \land \\
& \quad (\forall x : \text{nat}. \text{Terminates } x \land \text{True} \Rightarrow \text{Terminates } (x_1 x) \Rightarrow \text{True}) \Rightarrow \\
& \quad \text{Terminates } (\text{iter } x_1) \land \\
& \quad \forall x_2 : \text{nat}. \text{Terminates } x_2 \land \text{True} \Rightarrow \text{Terminates } (\text{iter } x_1 x_2) \land \\
& \quad \forall x_3 : \text{nat}. \text{Terminates } x_3 \land \text{True} \Rightarrow \text{Terminates } (\text{iter } x_1 x_2 x_3) \Rightarrow \text{True}
\end{align*}$$

**4.3 Translation of contexts**

Figure 11 gives a similar 2-part translation of typing contexts. The translation $\llbracket [\cdot] \rrbracket^C$ produces a simple-typing context $\Sigma$, while the translation $\llbracket [\cdot] \rrbracket^L$ produces a logical context $H$, which asserts, for each variable $x$, that $x$ terminates and has the property given by the $\llbracket [\cdot] \rrbracket^L$ 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 Appendix. Theorem 4 shows that the logical translation of types is sound: the property expressed by $\llbracket t \rrbracket^C_\theta$ can indeed be proved to hold for the translation $\llbracket t \rrbracket^C$ of terms of type $T$. 

**Theorem 3 (Soundness of Computational Translation).** If $\Gamma \vdash t : T \theta$, then $\llbracket \Gamma \rrbracket^C_\theta \vdash \llbracket t \rrbracket^C_\theta : \llbracket T \rrbracket^C_\theta$.

**Theorem 4 (Soundness of Logical Translation).** If $\Gamma \vdash t : T \theta$, then $\llbracket T \rrbracket^C_\theta \vdash \llbracket t \rrbracket^C_\theta : \llbracket T \rrbracket^C_\theta$.

---

Figure 11: Interpretation of contexts

\[
\begin{align*}
\llbracket [\cdot] \rrbracket^C & = \cdot \\
\llbracket [\cdot]\Gamma, x : T \rrbracket^C & = [\cdot], x : \llbracket T \rrbracket^C \\
\llbracket [\cdot]\Gamma, x : T \rrbracket^L & = [\cdot], \llbracket T \rrbracket^L \Gamma, x \ x
\end{align*}
\]
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 Teq programs can be fairly mechanically translated into programs using $(\cdot)^\nu$ by a translation similar to the one described by Wadler and Thiemann [18]. However, one interesting difference is that Teq 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 Teq extended with option types, and suppose we are given a decision procedure for equality of nat and a partial function which computes the minimum zero of a function:

\[
eqDec : \Pi x : \text{nat}. \Pi x' : \text{nat}. \text{Maybe}(x = x')
\]
\[
\text{minZero} : \Pi f : (\Pi x : \text{nat}. \text{nat}). \text{nat}
\]

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

\[
\lambda f . \lambda f'. eqDec (\text{minZero} f) (\text{minZero} f')
\]
\[
: \Pi f : (\Pi x : \text{nat}. \text{nat}). \Pi f' : (\Pi x : \text{nat}. \text{nat}). \text{Maybe}(\text{minZero} f = \text{minZero} f')
\]

However the naive translation of this into monadic form,

\[
\lambda f . \lambda f'. (\text{minZero} f) \gg= (\lambda m. (\text{minZero} f') \gg= (\lambda m'. \text{return}(eqDec m m'))),
\]

is not well typed, since the monadic bind \(\gg=\) : \(\forall A B. A^\nu \rightarrow (A \rightarrow B^\nu) \rightarrow B^\nu\) 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], \(\Omega\)MEGA [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

\(\text{Teq}\) 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 \(\text{Teq}\) in terms of a multi-sorted first-order theory of general-recursive functions. Future work includes further meta-theory, including type soundness for \(\text{Teq}\) 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.
A Proof of Theorem 3 (Soundness of Computational Translation)

The proof is a routine induction on the assumed $\tau^{\text{eql}}$ typing derivation, which we include here for thoroughness:
A.1 Case: \( T_{\text{VAR}} \)

\[
\frac{\Gamma(x) = T \quad \Gamma \vdash \text{Ok}}{\Gamma \vdash x : T \theta} \quad T_{\text{VAR}}
\]

This case follows directly from the easily proven fact that \( \Gamma(x) = T \) implies \( \llbracket \Gamma \rrbracket^C(x) = \llbracket T \rrbracket^C \).

A.2 Case: \( T_{\text{JOIN}} \)

\[
\frac{t \sim^* t_0 \quad t' \sim^* t_0}{\Gamma \vdash t : T ? \quad \Gamma \vdash t' : T' ? \quad \Gamma \vdash \text{join} : t = t' \theta} \quad T_{\text{JOIN}}
\]

The interpretation of the conclusion is just an instance of the \( \text{STY}_{\text{VAR}} \) rule. This is also true for the rules \( T_{\text{REIFY}}, T_{\text{INV}}, \) and \( T_{\text{CONTRA}} \), so we omit cases for those rules.

A.3 Case: \( T_{\text{CONV}} \)

\[
\frac{\Gamma \vdash t : [t_2 / x] T \theta \quad \Gamma \vdash t' : [t_1 / x] T}{\Gamma \vdash \text{join} : t = t' \theta} \quad T_{\text{CONV}}
\]

By the IH we have \( \llbracket \Gamma \rrbracket^C \vdash [t] \Gamma : \llbracket [t_2 / x] T \rrbracket^C \). We omit the straightforward proof that \( \llbracket [t_2 / x] T \rrbracket^C = \llbracket T \rrbracket^C = \llbracket [t_1 / x] T \rrbracket^C \), for any \( t_1, t_2, x, \) and \( T \). So the fact we have from the first premise is what is required for the conclusion. The case for \( T_{\text{REFLECT}} \) is similar, and so is omitted.

A.4 Case: \( T_{\text{ABS}} \)

\[
\frac{\Gamma, x : T' \vdash t : T \rho \quad \Gamma \vdash \Pi^\rho x : T'. T}{\Gamma \vdash \lambda x. t : \Pi^\rho x : T'. T \theta} \quad T_{\text{ABS}}
\]

By the IH we have \( \llbracket \Gamma \rrbracket^C \vdash [t] \Gamma : \llbracket T' \rrbracket^C \). This is equivalent to \( \llbracket \Gamma \rrbracket^C, x : \llbracket T' \rrbracket^C \vdash [t] \Gamma : \llbracket T' \rrbracket^C \), to which we can apply the simple typing rule \( \text{STY}_{\text{ABS}} \) to obtain \( \llbracket \Gamma \rrbracket^C \vdash \lambda x. [t] \Gamma : \llbracket T' \rrbracket^C \rightarrow \llbracket T \rrbracket^C \), which suffices by the definition of \( \llbracket [\cdot] \rrbracket^C \).

A.5 Case: \( T_{\text{APP}} \)

\[
\frac{\Gamma \vdash t : \Pi^\rho x : T'. T \theta \quad \Gamma \vdash t' : T' \theta \quad \rho \leq \theta}{\Gamma \vdash t t' : [t' / x] T \theta} \quad T_{\text{APP}}
\]

By the IH and the definition of \( \llbracket [\cdot] \rrbracket^C \), we have \( \llbracket \Gamma \rrbracket^C \vdash [t] \Gamma : \llbracket T' \rrbracket^C \rightarrow \llbracket T \rrbracket^C \) and also \( \llbracket \Gamma \rrbracket^C \vdash [t'] \Gamma : \llbracket T' \rrbracket^C \). We may apply the simple typing rule \( \text{STY}_{\text{APP}} \) to get \( \llbracket \Gamma \rrbracket^C \vdash [t] \Gamma : \llbracket T' \rrbracket^C \), which suffices, using again the definition of \( \llbracket [\cdot] \rrbracket^C \), and also the fact used above that \( \llbracket [t' / x] T \rrbracket^C = \llbracket T \rrbracket^C \).

A.6 Case: \( T_{\text{ZERO}} \)

\[
\frac{\Gamma \vdash \text{Ok}}{\Gamma \vdash 0 : \text{nat} \ \theta} \quad T_{\text{ZERO}}
\]

The desired conclusion is an instance of \( \text{STY}_{\text{ZERO}} \).
A.7 Case: T_SUC

\[ \Gamma \vdash t : \text{nat} \quad \theta \]
\[ \Gamma \vdash \text{Suc} t : \text{nat} \quad \theta \quad \text{T_SUC} \]

This case follows from the IH and then applying STY_SUC.

A.8 Case: T_REC_NAT

\[ p \not\in \text{fv} t \]
\[ \Gamma, f : \Pi x : \text{nat}. T, x : \text{nat}, p : \Pi x_1 : \text{nat}. \Pi f' : x = \text{Suc} x_1. \text{Terminates} \quad (f x_1) \vdash t : T \quad \downarrow \quad \text{T_REC_NAT} \]

By the IH, we have:

\[ [[\Gamma, f : \Pi x : \text{nat}. T, x : \text{nat}, p : \Pi x_1 : \text{nat}. \Pi f' : x = \text{Suc} x_1. \text{Terminates} \quad (f x_1)]]^C \vdash [[t]]^C : [[T]]^C \]

This is equivalent to:

\[ [[\Gamma]]^C, f : \text{nat} \rightarrow [[T]]^C, x : \text{nat}, p : \text{nat} \rightarrow \text{nat} \vdash [[t]]^C : [[T]]^C \]

We omit the straightforward proof that \( \text{fv}[[t]]^C = \text{fv} t \), which gives us \( p \not\in \text{fv}[[t]]^C \). We also omit the straightforward proof of Strengthening for our simply typed system, which says \( \Sigma, x : A \vdash t : A' \) implies \( \Sigma \vdash t : A \) if \( x \not\in \text{fv} t \). Using this Strengthening property for the simply typed system, we then have:

\[ [[\Gamma]]^C, f : \text{nat} \rightarrow [[T]]^C, x : \text{nat} \vdash [[t]]^C : [[T]]^C \]

We may now just apply the rule STY_REC, to conclude the desired \( [[\Gamma]]^C \vdash \text{rec} f(x) = [[t]]^C : [[T]]^C \).

The case for T_REC is the same as the last part of this case, and so is omitted.

A.9 Case: T_CASE_NAT

\[ \Gamma \vdash t : \text{nat} \quad \theta \]
\[ \Gamma \vdash t' : [0 / x] T \quad \theta \]
\[ \Gamma \vdash t'' : \Pi x' : \text{nat}. \text{Suc} x' / x \quad \text{T \quad \rho \leq \theta} \]
\[ \Gamma \vdash \text{case} t t' t'' : [t / x] T \quad \theta \quad \text{T_CASE_NAT} \]

By the IH and the definition of \( [[\cdot]]^C \), we have:

- \( [[\Gamma]]^C \vdash [[t]]^C : \text{nat} \)
- \( [[\Gamma]]^C \vdash [[t']]^C : [[(0 / x)T]]^C \)
- \( [[\Gamma]]^C \vdash [[t'']]^C : \text{nat} \rightarrow [[\text{Suc} x'/x]T]^C \)

Using again the property mentioned above, that \( [[t / x] T]^C = [[T]]^C \), we may then apply the rule STY_CASE_NAT to obtain the desired conclusion.

A.10 Case: T_ABORT

\[ \Gamma \vdash \text{Ok} \quad \text{T_ABORT} \]

The desired conclusion is just an instance of STY_ABORT.
B Proof of Theorem 4 (Soundness of Logical Translation)

We prove this by induction on the structure of the assumed derivation. Note first that if the interpretation of a $\text{T}_{\text{eq}} \downarrow$ typing judgment with effect $\downarrow$ holds – that is, if we have $\llbracket \Gamma \rrbracket^C; \llbracket \Gamma \rrbracket^L \vdash \text{Terminates} \llbracket \tau \rrbracket^C \land \llbracket T \rrbracket^L \llbracket \tau \rrbracket^C$ – then we also have $\llbracket \Gamma \rrbracket^C; \llbracket \Gamma \rrbracket^L \vdash \text{Terminates} \llbracket \tau \rrbracket^C \Rightarrow \llbracket T \rrbracket^L \llbracket \tau \rrbracket^C$, which is the interpretation of the similar $\text{T}_{\text{eq}} \downarrow$ typing judgment with effect $?$. So in cases where we can prove the interpretation of the judgment with $\downarrow$, we can omit the proof of the interpretation of the judgment with $?$.

B.1 Case: T_VAR

\[
\frac{\Gamma(x) = T \quad \Gamma \vdash \text{Ok}}{\Gamma \vdash x : T \theta} \quad \text{T_VAR}
\]

This case follows directly from the fact that the logical interpretation of the $\text{T}_{\text{eq}} \downarrow$ typing context $\Gamma$ must contain $\text{Terminates} x$ and $\llbracket T \rrbracket^L x$, since $\Gamma(x) = T$.

B.2 Case: T_JOIN

\[
\frac{t \approx t_0 \quad t' \approx t_0}{\Gamma \vdash t : T \Rightarrow \Gamma \vdash t' : T' \Rightarrow} \quad \text{T_JOIN}
\]

From the fact that $t \approx t_0$ implies $\llbracket t \rrbracket^C \sim^* \llbracket t_0 \rrbracket^C$ (we omit the easy proof), we have that $\llbracket t \rrbracket^C$ and $\llbracket t' \rrbracket^C$ are joinable. Our equational theory allows us to prove that joinable terms are equal (regardless of whether they are terminating or not). Hence, we can indeed prove the logical interpretation of the type in the conclusion, namely $\llbracket t \rrbracket^C = \llbracket t' \rrbracket^C$.

B.3 Case: T_CONV

\[
\frac{\Gamma \vdash t : \llbracket t_2 / x \rrbracket T \theta}{\llbracket t \rrbracket^C \vdash \llbracket t \rrbracket^C \Rightarrow \llbracket t_1 / x \rrbracket \llbracket T \rrbracket^L \theta} \quad \text{T_CONV}
\]

By the IH we have:

$\llbracket \Gamma \rrbracket^C; \llbracket \Gamma \rrbracket^L \vdash \llbracket t_2 / x \rrbracket \llbracket T \rrbracket^L \theta \llbracket t \rrbracket^C$

We omit the straightforward proof that

$\llbracket \llbracket t_2 / x \rrbracket T \rrbracket^L \theta \llbracket t \rrbracket^C = \llbracket \llbracket t_2 \rrbracket^C / x \rrbracket (\llbracket T \rrbracket^L \theta \llbracket t \rrbracket^C)$

Using this fact, it suffices to prove the similar statement, except with $\llbracket t_1 \rrbracket^C$ in place of $\llbracket t_2 \rrbracket^C$. But this follows by PV\_SUBST, using the fact that $\llbracket \Gamma \rrbracket^C; \llbracket \Gamma \rrbracket^L \vdash \llbracket t_1 \rrbracket^C = \llbracket t_2 \rrbracket^C$. We have this from the formula we get by the IH for the second premise, noting that

$\llbracket t_1 = t_2 \rrbracket^L \llbracket t' \rrbracket^C = \text{Terminates} \llbracket t' \rrbracket^C \land \llbracket t_1 \rrbracket^C = \llbracket t_2 \rrbracket^C$
B.4 Case: T\_REFLECT

\[
\frac{\Gamma \vdash t : T; \quad \Gamma \vdash t' : \text{Terminates} \quad t \downarrow}{\Gamma \vdash t : T \theta} \quad \text{T\_REFLECT}
\]

If $\theta = \theta'$, the desired result follows immediately from the IH applied to the first premise. If $\theta \neq \theta'$ but $\theta = \downarrow$, we have already observed above that we can obtain the logical translation of a $T^{=\downarrow}$ typing judgment with effect $?\downarrow$ if we have the similar translation with effect $\downarrow$. So it suffices to consider just the case where $\theta = ?$ but $\theta' = \downarrow$. By the IH for the second premise, we have:

\[
\llbracket \Gamma \rrbracket^C; \llbracket \Gamma \rrbracket^L \vdash \text{Terminates} \llbracket t' \rrbracket^C \land \text{Terminates} \llbracket t \rrbracket^C
\]

The second conjunct of this is exactly what we need to obtain the desired conclusion from what we get from the IH applied to the first premise, which is:

\[
\llbracket \Gamma \rrbracket^C; \llbracket \Gamma \rrbracket^L \vdash \text{Terminates} \llbracket t \rrbracket^C \Rightarrow \llbracket T \rrbracket^L \llbracket t \rrbracket^C
\]

B.5 Case: T\_REIFY

\[
\frac{\Gamma \vdash t : T \downarrow}{\Gamma \vdash \text{terminates} : \text{Terminates} \quad t \theta} \quad \text{T\_REIFY}
\]

The IH for the premise is:

\[
\llbracket \Gamma \rrbracket^C; \llbracket \Gamma \rrbracket^L \vdash \text{Terminates} \llbracket t \rrbracket^C \land \llbracket T \rrbracket^L \llbracket t \rrbracket^C
\]

From this by PV\_ANDE1 we obtain the translation of the conclusion, using also the axiom PV\_TERMINATES0 (to show Terminates $\llbracket \text{terminates} \rrbracket^C$).

B.6 Case: T\_CTXTERM

\[
\frac{\Gamma \vdash t : \text{Terminates} \quad \llbracket \cdot \rrbracket \theta \quad \Gamma \vdash \llbracket \cdot \rrbracket \theta}{\Gamma \vdash t : \text{Terminates} \quad t \theta} \quad \text{T\_CTXTERM}
\]

It is sufficient to show Terminates $\llbracket \cdot \rrbracket^C \llbracket t' \rrbracket^C \Rightarrow \text{Terminates} \llbracket t \rrbracket^C$, where $\llbracket \cdot \rrbracket^C$ is the context determined by the obvious extension of $\llbracket \cdot \rrbracket$ from terms to contexts. This formula easily follows using PV\_TERMINATE.

B.7 Case: T\_ABS

\[
\frac{\Gamma, x : T' \vdash t : T \rho \quad \Gamma \vdash \Pi^\rho x : T'.T}{\Gamma \vdash \lambda x. t : \Pi^\rho x : T'.T \theta} \quad \text{T\_ABS}
\]

Applying the IH to the first premise gives us:

\[
\llbracket \Gamma \rrbracket^C, x : \llbracket T' \rrbracket^C; \llbracket \Gamma \rrbracket^L, \llbracket T' \rrbracket^L ; x \vdash \llbracket T \rrbracket^L \llbracket t \rrbracket^C
\]

As remarked above, it suffices to prove the conclusion for when $\theta' = \downarrow$, since this implies the case when $\theta' = ?$. So we must prove:

\[
\llbracket \Gamma \rrbracket^C; \llbracket \Gamma \rrbracket^L \vdash \text{Terminates} \quad \lambda x. t \land \forall x : \llbracket T' \rrbracket^C. \llbracket T' \rrbracket^L ; x \Rightarrow \llbracket T \rrbracket^L \llbracket (\lambda x. t) \rrbracket^C
\]

The first conjunct is provable by PV\_TERMABS. The second follows easily from the fact we obtained from the IH, by applying PV\_SUBST with the equation $t = (\lambda x. t)x$, which holds by PV\_OPSEM (and then using also PV\_ALLI and PV\_IMPI).
B.8 Case: $T_{\text{App}}$

\[
\Gamma \vdash t : \Pi \rho \cdot T \cdot \theta \quad \Gamma \vdash t' : T' \cdot \theta \quad \rho \leq \theta \quad T_{\text{App}}
\]

We first case-split on whether $\theta = ?$ or $\theta = \bot$. If $\theta = ?$, then by the IH, we have:

- $[[\Gamma]]^C; [[\Gamma]]^L \vdash \text{Terminates}[[t]]^C \Rightarrow \forall x : [[T']]^C. [[T']]^L_x \Rightarrow [[T']]^L_\rho [[t]]^C x$

- $[[\Gamma]]^C; [[\Gamma]]^L \vdash \text{Terminates}[[t']]^C \Rightarrow [[T']]^L [[t']]^C$

We must prove:

\[
[[\Gamma]]^C; [[\Gamma]]^L \vdash \text{Terminates}[[t']]^C \Rightarrow [[t' / x] T]^L [[t']]^C
\]

So (using PV_IMP) assume $\text{Terminates}[[t']]^C$, and prove $[[[t' / x] T]^L [[t']]^C$. By PV_TERMINV, we have $\text{Terminates}[[t]]^C$ and $\text{Terminates}[[t']]^C$. So from the facts we obtained above by the IH, we get (using PV_IMPE):

- $[[\Gamma]]^C; [[\Gamma]]^L \vdash \forall x : [[T']]^C. [[T']]^L_x \Rightarrow [[T']]^L_\rho [[t]]^C x$

- $[[\Gamma]]^C; [[\Gamma]]^L \vdash [[T']]^L [[t']]^C$

We can instantiate the quantifier in the first fact, using PV_ALLE and Theorem 3 (to get $[[\Gamma]]^C \vdash [[t']]^C : [[T']]^C$). This gives us the following from the first fact:

\[
[[\Gamma]]^C; [[\Gamma]]^L \vdash [[T']]^L_x \Rightarrow [[[t']^C / x] [T]^L_\rho [[t]]^C [[t']]^C
\]

The antecedent of this implication is provable from the second fact above and $\text{Terminates}[[t']]^C$, giving us:

\[
[[\Gamma]]^C; [[\Gamma]]^L \vdash [[[t']^C / x] [T]^L_\rho [[t]]^C [[t']]^C
\]

Since we already have $\text{Terminates}[[t]]^C$ $[[t']]^C$, from this we obtain (no matter what the value of $\rho$ is)

\[
[[\Gamma]]^C; [[\Gamma]]^L \vdash [[[t']^C / x] [T]^L_\rho [[t]]^C [[t']]^C
\]

The desired conclusion then follows from the fact that $[[[t']^C / x] [T]^L_\rho = [[[t'] / x] T]^L_\rho$. We omit the straightforward proof of this fact.

Now we must consider the case where $\theta = \bot$, and hence $\rho = \bot$ (from the rule’s third premise). In this case, the IH gives us:

- $[[\Gamma]]^C; [[\Gamma]]^L \vdash \text{Terminates}[[t]]^C \land \forall x : [[T']]^C. [[T']]^L_x \Rightarrow [[T']]^L_\rho [[t]]^C x$

- $[[\Gamma]]^C; [[\Gamma]]^L \vdash \text{Terminates}[[t']]^C \land [[T']]^L [[t']]^C$

We must prove:

\[
[[\Gamma]]^C; [[\Gamma]]^L \vdash [[[t'] / x] T]^L_\rho [[t']]^C
\]

By the same reasoning as in the previous case, we obtain:

\[
[[\Gamma]]^C; [[\Gamma]]^L \vdash [[[t'] / x] T]^L_\rho [[t]]^C [[t']]^C
\]

But this is exactly what we must prove, since $\rho = \bot$. 

22
### B.9 Case: T.ZERO

\[
\frac{\Gamma \vdash \text{Ok}}{\Gamma \vdash 0 : \text{nat} \ \theta} \quad \text{T.ZERO}
\]

It suffices to prove \([\Gamma]^C; [\Gamma]^L \vdash \text{Terminates} \ 0 \land \text{True}\), which follows easily using PV.TERM0 and PV.TRUEI.

### B.10 Case: T.SUC

\[
\frac{\Gamma \vdash t : \text{nat} \ \theta}{\Gamma \vdash \text{Succ} \, t : \text{nat} \ \theta} \quad \text{T.SUC}
\]

We again case-split on whether \(\theta = ?\) or \(\theta = \bot\). In the former case, the IH gives us:

\([\Gamma]^C; [\Gamma]^L \vdash \text{Terminates} \ [t]^C \land \text{True}\)

We must prove:

\([\Gamma]^C; [\Gamma]^L \vdash \text{Terminates} \ S \ [t]^C \land \text{True}\)

This follows by PV.TERMINATES. If \(\theta = \bot\), we must prove:

\([\Gamma]^C; [\Gamma]^L \vdash \text{Terminates} \ [t]^C \Rightarrow \text{True}\)

But this is holds just by PV.IMPI and PV.TRUEI.

### B.11 Case: T.REC

\[
\frac{\Gamma, f : \Pi^x:T'.T, x : T' \vdash t : T \ ?}{\Gamma \vdash \text{rec} \, f(x) = t : \Pi^x:T'.T \ \theta} \quad \text{T.REC}
\]

By the IH, we have:

\([\Gamma]^C, f : [T']^C \rightarrow [T']^C, x : [T']^C ; [\Gamma]^L, \forall x : [T']^C. \ [T']^L \ x \Rightarrow [T']^L \ f (x), [T']^L \ x \vdash [T]^L \ [t]^C\)

Applying PV.IMPI and PV.ALLI, we get:

\([\Gamma]^C, f : [T']^C \rightarrow [T']^C ; [\Gamma]^L, \forall x : [T']^C. \ [T']^L \ x \Rightarrow [T']^L \ f (x) \vdash \forall x : [T']^C. \ [T']^L \ x \Rightarrow [T]^L \ [t]^C\)

This exactly matches the logical premise of the PV.COMPIND rule, with \(F\) taken to be \(\forall x : [T']^C. \ [T']^L \ x \Rightarrow [T]^L \ [t]\)

\[
\Sigma, f : A' \rightarrow A ; H, \forall x : A'. [f x/z] F \vdash \forall x : A'. [t/z] F \quad \Sigma \vdash \text{rec} \, f(x) = t : A' \rightarrow A \\
\Sigma ; H \vdash \forall x : A'. \text{Terminates} \ (\text{rec} \, f(x) = t) x \Rightarrow [\text{(rec} \, f(x) = t) x/z] F \quad \text{PV.COMPIND}
\]

So applying PV.COMPIND, we get the following fact (call it (J)):

\([\Gamma]^C ; [\Gamma]^L \vdash \forall x : [T']^C. \text{Terminates} \ (\text{rec} \, f(x) = [t]^C) x \Rightarrow [T']^L \ x \Rightarrow [T]^L \ (\text{rec} \, f(x) = [t]^C) x\)

Note that the consequent \([T]^L \ (\text{rec} \, f(x) = [t]^C) x\) of this implication is, by definition of \([\cdot]^L\):

\(\text{Terminates} \ (\text{rec} \, f(x) = [t]^C) x \Rightarrow [T]^L \ (\text{rec} \, f(x) = [t]^C) x\)
So the premise of the implication in (J) is redundant, and we can deduce the following (J’):
\[
[[\Gamma]^C; [[\Gamma]^L] \vdash \forall x: [[T']^C]. \quad [[T']^L_f x \Rightarrow [[T]^L_f (\text{rec } f(x) = [[t]^C) x
\]

It suffices now to prove:
\[
[[\Gamma]^C; [[\Gamma]^L] \vdash \text{TermNat } \text{rec } f(x) = [[t]^C \land \forall x: [[T']^C]. \quad [[T']^L_f x \Rightarrow [[T]^L_f (\text{rec } f(x) = [[t]^C) x
\]

The first conjunct is provable by PV_TERMREC. The second now follows directly by what we have just deduced above.

**B.12 Case: T_REC_NAT**

\[
p \notin \text{fv } \Gamma, f : \Pi x: \text{nat}. T, x : \text{nat}, p : \Pi x_1 : \text{nat}. \Pi p' : x = \text{Suc } x_1. \text{TermNat } (f x_1) \vdash t : T \quad \text{T_REC_NAT}
\]

By the IH, we have
\[
[[\Gamma]^C, f : \text{nat} \rightarrow [[T]^C, x : \text{nat}, p : \Pi x_1 : \text{nat}. (\Pi p' : x = \text{Suc } x_1) \text{TermNat } (f x_1) \vdash t : T \quad \text{T_REC_NAT}
\]

where:
\[
F_1 = \forall x_1 : \text{nat}. \text{TermNat } x_1 \land \text{True} \Rightarrow [[T]^L_f (f x_1)
\]
\[
F_2 = \forall x_1 : \text{nat}. \text{TermNat } x_1 \land \text{True} \Rightarrow \text{TermNat } (p x_1) \land \exists x_2 : \text{nat}. \text{TermNat } x_2 \land x = \text{Suc } x_1 \Rightarrow \text{TermNat } ((p x_1) x_2) \land
\]

We may easily show that we can replace $F_1$ and $F_2$ by the following simplified versions:
\[
F'_1 = \forall x_1 : \text{nat}. \text{TermNat } x_1 \Rightarrow [[T]^L_f (f x_1)
\]
\[
F'_2 = \forall x_1 : \text{nat}. \text{TermNat } x_1 \Rightarrow x = \text{Suc } x_1 \Rightarrow \text{TermNat } (f x_1)
\]

This (and similar simplifications), followed by some uses of PV_ALL1 and PV_IMPI, and also supplying an arbitrary lambda-abstraction of type $\text{nat} \rightarrow (\text{nat} \rightarrow \text{nat})$ for $p$ gives us the following central assumption (call it (J)) from the judgment above:
\[
[[\Gamma]^C; [[\Gamma]^L] \vdash \forall x : \text{nat}. \text{TermNat } x \Rightarrow \forall f : \text{nat} \rightarrow [[T]^C. (\text{TermNat } f \land F'_1 \land F'_2) \Rightarrow [[T]^L_f ([t]^C
\]

It suffices to show:
\[
[[\Gamma]^C; [[\Gamma]^L] \vdash \text{TermNat } \text{rec } f(x) = [[t]^C \land \forall x : \text{nat}. \text{TermNat } x \land \text{True} \Rightarrow [[T]^L_f (\text{rec } f(x) = [[t]^C) x
\]

We have the first conjunct by PV_TERMREC. For the second, it suffices to show:
\[
[[\Gamma]^C; [[\Gamma]^L] \vdash \forall x : \text{nat}. \text{TermNat } x \Rightarrow [[T]^L_f (\text{rec } f(x) = [[t]^C) x
\]

We do this by induction (using PV_IND). First, though, we observe that the reasoning we used in the previous case (for T_REC) to prove what we called (J’) applies here (except that here we have some additional assumptions in the context). This lets us deduce the following, which we will call (J’) (essentially the (J’) from the previous case, with $T'$ replaced by $\text{nat}$):
\[
\forall x : \text{nat}. \text{TermNat } x \Rightarrow [[T]^L_f (\text{rec } f(x) = [[t]^C) x
\]
So now for the base case, we must prove:

$$[[\Gamma]^C; [\Gamma]^L] \vdash [[0/x] T]^L_1 (\text{rec } f(x) = [r]^C) \ 0$$

This follows easily using PV\_SUBST and PV\_OPSEM from:

$$[[\Gamma]^C; [\Gamma]^L] \vdash [[0/x] T]^L_1 ([0/x][\text{rec } f(x) = [r]^C / [r]^C]$$

We obtain this by instantiating (J) above with 0 for x, and \(\text{rec } f(x) = [r]^C\) for f. We have the required proofs of termination by PV\_TERM0 and PV\_TERMRec. We have a proof of the appropriately instantiated premise \(F'_1\) of (J), since this is exactly (J\'). We easily prove the instantiation of premise \(F'_2\) of (J), since this is:

$$\forall x_1 : \text{nat. Terminates } x_1 \Rightarrow 0 = \text{Suc } x_1 \Rightarrow \text{Terminates } ((\text{rec } f(x) = [r]^C) x_1)$$

This formula is easily proved using PV\_CONTRA with premise \(0 = \text{Suc } x_1\). So from (J), with these instantiations and proven premises, we obtain the following, which is exactly what we had to prove:

$$[[\Gamma]^C; [\Gamma]^L] \vdash [[0/x] T]^L_1 (\text{rec } f(x) = [r]^C) \ 0$$

For the step case, we must now prove:

$$[[\Gamma]^C, x' : \text{nat}; [\Gamma]^L, \text{Terminates } x', [[x'/x] T]^L_1 (\text{rec } f(x) = [r]^C) x' \vdash [[\text{Suc } x'/x] T]^L_1 (\text{rec } f(x) = [r]^C) (S \ x')$$

Now we instantiate (J) above with Suc\(x'\) for x, and again \(\text{rec } f(x) = [r]^C\) for f. We easily obtain the required proofs of termination. The instantiated \(F'_1\) we again have by (J\'). The instantiated premise \(F'_2\) is:

$$\forall x_1 : \text{nat. Terminates } x_1 \Rightarrow \text{Suc } x' = \text{Suc } x_1 \Rightarrow \text{Terminates } ((\text{rec } f(x) = [r]^C) x_1)$$

We can prove this premise as follows. Assume arbitrary terminating \(x_1\) of sort nat, and assume Suc\(x' = \text{Suc } x_1\). Using PV\_SUBST and PV\_OPSEM, we can derive \(x' = x_1\) from this:

$$\frac{\text{Suc } x' = \text{Suc } x_1 \quad \text{case } (\text{Suc } x') 0 \lambda z. z = x' \quad \text{PV\_OPSEM} \quad \text{PV\_SUBST}}{\text{case } (\text{Suc } x_1) 0 \lambda z. z = x' \quad \text{PV\_OPSEM} \quad \text{PV\_SUBST}}$$

So now to complete the proof of the instantiated premise \(F'_2\), we need only prove

$$\text{Terminates } (\text{rec } f(x) = [r]^C) x'$$

But this follows directly from the assumption we have in this step case of PV\_IND:

$$[[x'/x] T]^L_1 (\text{rec } f(x) = [r]^C) x'$$

So we have all the premises required by (J), and we can prove the following (applying a derived weakening rule, whose easy proof is omitted, to (J) to add our other assumptions to its contexts):

$$[[\text{Suc } x'/x] T]^L_1 \ [\text{Suc } x'/x][\text{rec } f(x) = [r]^C / [r]^C$$

This now easily implies the required conclusion, using PV\_SUBST with the following equation, which holds by PV\_OPSEM:

$$[\text{Suc } x'/x][\text{rec } f(x) = [r]^C / [r]^C] = (\text{rec } f(x) = [r]^C)(\text{Suc } x')$$
B.13 Case: T_CASENAT

\[ \Gamma \vdash t : \text{nat} \quad \Gamma \vdash t' : [0/x] T \theta \\
\Gamma \vdash t'' : \Pi P.x' : \text{nat} . [\text{Suc} x'/x] T \theta \quad \rho \leq \theta \]

\[ \Gamma \vdash \text{case } t t' t'' : \lbrack t/x \rbrack T \theta \]

As for some cases above, we begin by case-splitting on whether \( \theta = ? \) or \( \theta = \lbrack \rbrack \). Suppose \( \theta = ? \). Then applying the IH to the second and third premises, and then a few basic logical simplifications, gives us:

1. \( [[\Gamma]]^C ; [[\Gamma]]^L \vdash \text{Terminates} \lbrack t' \rbrack^C \Rightarrow [[0/x] T]^L [[t']^C \]
2. \( [[\Gamma]]^C ; [[\Gamma]]^L \vdash \text{Terminates} \lbrack t'' \rbrack^C \Rightarrow \forall x' : \text{nat} . \text{Terminates} x' \Rightarrow [[\text{Suc} x'/x] T]^L \rho ([[t'']^C x']) \)

We must show:

\[ [[\Gamma]]^C ; [[\Gamma]]^L \vdash \text{Terminates} \lbrack \text{case } t t' t'' \rbrack^C \Rightarrow [[t/x] T]^L [[\text{case } t t' t'']^C \]

So assume \( \text{Terminates} \lbrack \text{case } t t' t'' \rbrack^C \), and show \( [[t/x] T]^L [[\text{case } t t' t'']^C \). By PV\_TERMINV, this assumption implies \( \text{Terminates} \lbrack t \rbrack^C \). Since \( [[\Gamma]]^C \vdash [t] : \text{nat} \) by Theorem 3, we will now seek to prove the following by PV\_IND:

\[ [[\Gamma]]^C ; [[\Gamma]]^L \vdash \forall x : \text{nat} \cdot \text{Terminates } x \Rightarrow \text{Terminates} \lbrack \text{case } x t' t'' \rbrack^C \Rightarrow [[T]^L [[\text{case } x t' t'']^C \]

If we can derive this judgment, then we can instantiate \( x \) with \( [t]^C \) (for which we have \( \text{Terminates} \lbrack t \rbrack^C \)) to conclude the desired

\[ [[\Gamma]]^C ; [[\Gamma]]^L \vdash \text{Terminates} \lbrack \text{case } t t' t'' \rbrack^C \Rightarrow [[t/x] T]^L [[\text{case } t t' t'']^C \]

To apply PV\_IND as desired, we must prove the base case and step case of the induction:

- \( [[\Gamma]]^C ; [[\Gamma]]^L \vdash \text{Terminates} \lbrack \text{case } 0 t' t'' \rbrack^C \Rightarrow [[0/x] T]^L [[\text{case } 0 t' t'']^C \)
- \( [[\Gamma]]^C , x' : \text{nat} ; [[\Gamma]]^L \vdash \text{Terminates} x' \Rightarrow \text{Terminates} \lbrack \text{case } (\text{Suc} x') t' t'' \rbrack^C \Rightarrow [[\text{Suc} x'/x] T]^L [[\text{case } (\text{Suc} x') t' t'']^C \)

This case base follows from fact (1) above, using the equation \( [[\text{case } 0 t' t'']^C = [[t']^C \). This equation is easily shown by the definition of \( [.]^C \) and PV\_OPSEM. So we now prove the step case. First, we simplify the desired formula using the easily proved equation \( [[\text{case } (\text{Suc} x') t' t'']^C = [[t'']^C x'] \). So our new goal formula is

\[ [[\Gamma]]^C , x' : \text{nat} ; [[\Gamma]]^L , \text{Terminates} x' \Rightarrow \text{Terminates} [[t'' x']^C \Rightarrow [[\text{Suc} x'/x] T]^L [[t'' x']^C \]

So assume \( \text{Terminates } x' \) and \( \text{Terminates} [[t'' x']^C \), and show \( [[\text{Suc} x'/x] T]^L [[t'' x']^C \). Instantiating fact (2) above with \( x' \) and these assumptions, we get:

\[ [[\text{Suc} x'/x] T]^L \rho ([[t'']^C x')] \]

This implies the desired formula in either possible case for \( \rho \).

Now let us assume \( \theta = \lbrack \rbrack \). This case is similar to the above, so we give fewer details. The IH for the three premises gives us:

1. \( [[\Gamma]]^C ; [[\Gamma]]^L \vdash \text{Terminates} [[t]^C \land \text{True} \)
2. \( [[\Gamma]]^C ; [[\Gamma]]^L \vdash \text{Terminates} [[t']^C \land [[0/x] T]^L [[t']^C \)

26
3. \([\Gamma]^C; \Gamma]^L \vdash \text{Terminates} \[t''\]^C \land \forall x' : \text{nat} \text{. Terminates} x' \Rightarrow \[[\text{Suc} x'/x] T\]_\rho^L (\[t''\]^C x')

We must show
\[[\Gamma]^C; \Gamma]^L \vdash \text{Terminates} (\text{case } t t'l''\]^C \land \[\text{[t / x]} T\]^L (\text{case } t t'l''\]^C

Since we have \text{Terminates} \[t\]^C and \[[\Gamma]^C \vdash [t]^C : \text{nat}, it suffices to prove the following more general statement:
\[[\Gamma]^C; \Gamma]^L \vdash \forall x : \text{nat} \text{. Terminates} (\text{case } x t'l''\]^C \land \[\text{T}\]^L (\text{case } x t'l''\]^C

We again apply PV.IND. The base case is again immediate using \[[\text{case } 0 t'l''\]^C = \[t'\]^C. Similarly reasoning as for the step case above gives us:
\[[\text{Suc} x'/x] T\]_\rho^L (\[t''\]^C x')

Since \(\rho\) must be \(\downarrow\) in this case, we obtain from this fact the desired \text{Terminates} (\[t''\]^C x'), as well as
\[[\text{Suc} x'/x] T\]_\rho^L (\[t''\]^C x')

B.14 Case: T\_CONTRA
\[
\Gamma \vdash t : 0 = \text{Suc} t' \downarrow \quad \text{T\_CONTRA}
\]
By the IH we have \[[\Gamma]^C; \Gamma]^L \vdash \text{Terminates} \[t\]^C \land 0 = S \[t'\]^C. We must show \[[\Gamma]^C; \Gamma]^L \vdash \[T\]_\theta^L 0.
But this fact follows directly from the second conjunct of the fact we have, using PV\_CONTRA.

B.15 Case: T\_ABORT
\[
\Gamma \vdash \text{Ok} \quad \text{T\_ABORT}
\]
We must prove \[[\Gamma]^C; \Gamma]^L \vdash \text{Terminates abort} \Rightarrow \[T\]_\theta^L \text{ abort}. But this follows directly by PV\_IMPI from PV\_NOTTERMINATESABORT.

C Typing rules for system W'

\[
\Sigma \vdash t : A \quad \text{Simple-type assignment}
\]
\[
\Sigma(x) = A \quad \text{STY\_VAR}
\]
\[
\Sigma, x : A_1 \vdash t : A_2 \quad \text{STY\_ABS}
\]
\[
\Sigma \vdash \lambda x.t : A_1 \rightarrow A_2 \quad \text{STY\_APP}
\]
\[
\Sigma \vdash 0 : \text{nat} \quad \text{STY\_ZERO}
\]
\[
\Sigma \vdash t : \text{nat} \quad \text{STY\_SUC}
\]

27
\[
\begin{align*}
\frac{\Sigma, f : \text{nat} \to A, x : \text{nat} \vdash t : A}{\Sigma \vdash \text{rec } f(x) = t : \text{nat} \to A} & \quad \text{STY} \text{REC} \\
\frac{\Sigma \vdash t : \text{nat} \quad \Sigma \vdash t' : A \quad \Sigma \vdash t'' : \text{nat} \to A}{\Sigma \vdash \text{case } t \mid t' \mid t'' : A} & \quad \text{STY} \text{CASENAT} \\
\frac{\Sigma \vdash \text{abort} : A}{\Sigma \vdash \text{abort} : A} & \quad \text{STY} \text{ABORT}
\end{align*}
\]