Equality, Quasi-Implicit Products, and Large Eliminations

Vilhelm Sjöberg
Computer and Information Science
University of Pennsylvania
vilhelm@cis.upenn.edu

Aaron Stump
Computer Science
The University of Iowa
astump@acm.org

Abstract

This paper presents a type theory with a form of equality reflection: provable equalities can be used to coerce the type of a term. Coercions and other annotations, including implicit arguments, are dropped during reduction of terms. We develop the metatheory for an undecidable version of the system with unannotated terms. We then devise a decidable system with annotated terms, justified in terms of the unannotated system. Finally, we show how the approach can be extended to account for large eliminations, using what we call quasi-implicit products.

1 Introduction

The main goal of this paper, as of several recent works, is to facilitate external reasoning about dependently typed programs [9, 2]. This is hampered if one must reason about specificational data occurring in terms. Specificational data are data which have no effect on the result of the computation, and are present in program text solely for verification purposes. In traditional formal methods, specification data are also sometimes called ghost data. For example, consider the familiar example of vectors $\langle \texttt{vec}\ \phi\ l \rangle$ indexed by both the type ϕ of the elements and the length l of the vector. An example dependently typed program is the append_ϕ function (we work here with monomorphic functions, but will elide type subscripts), operating on vectors holding data of type ϕ. We can define *append* so that it has the following type, assuming a standard definition of *plus* on unary natural numbers $\texttt{nat}$:

$$\mathit{append}\ :\ \Pi l_1 : \texttt{nat}.\Pi l_2 : \texttt{nat}.\Pi v_1 : \langle \texttt{vec}\ \phi\ l_1 \rangle.\Pi v_2 : \langle \texttt{vec}\ \phi\ l_2 \rangle.\ \langle \texttt{vec}\ \phi\ (\mathit{plus}\ l_1\ l_2) \rangle$$

We might wish to prove that *append* is associative. In type theories such as Coq's Calculus of Inductive Constructions, we would do this by showing that the following type is inhabited:

$$\begin{array}{l}\Pi l_1 : \texttt{nat}.\Pi l_2 : \texttt{nat}.\Pi l_3 : \texttt{nat}.\Pi v_1 : \langle \texttt{vec}\ \phi\ l_1 \rangle.\Pi v_2 : \langle \texttt{vec}\ \phi\ l_2 \rangle.\Pi v_3 : \langle \texttt{vec}\ \phi\ l_3 \rangle.\\ \quad (\mathit{append}\ (\mathit{plus}\ l_1\ l_2)\ l_3\ (\mathit{append}\ l_1\ l_2\ v_1\ v_2)\ v_3) = (\mathit{append}\ l_1\ (\mathit{plus}\ l_2\ l_3)\ v_1\ (\mathit{append}\ l_2\ l_3\ v_2\ v_3))\end{array}$$

Notice how the lengths of the vectors are cluttering even the statement of this theorem. Tools like Coq allow such arguments to be elided, when they can be uniquely reconstructed. So the theorem to prove can be written in the much more palatable form:

$$\begin{array}{l}\Pi l_1 : \texttt{nat}.\Pi l_2 : \texttt{nat}.\Pi l_3 : \texttt{nat}.\Pi v_1 : \langle \texttt{vec}\ \phi\ l_1 \rangle.\Pi v_2 : \langle \texttt{vec}\ \phi\ l_2 \rangle.\Pi v_3 : \langle \texttt{vec}\ \phi\ l_3 \rangle.\\ \quad (\mathit{append}\ (\mathit{append}\ v_1\ v_2)\ v_3) = (\mathit{append}\ v_1\ (\mathit{append}\ v_2\ v_3))\end{array}$$

This is much more readable. But as others have noted, while the indices have been elided, they are not truly erased. This means that the proof of associativity of *append* must make use of associativity also of *plus*, in order for the lengths of the two vectors (on the two sides of the equation) to be equal. Indeed, even stating this equation may require some care, since the types of the two sides are not definitionally equal: one has $(\mathit{plus}\ (\mathit{plus}\ l_1\ l_2)\ l_3)$ where the other has $(\mathit{plus}\ l_1\ (\mathit{plus}\ l_2\ l_3))$. This is where techniques like heterogeneous equality come into play [7].

One solution to this problem is via intersection types, also called in this setting *implicit products*, as in the Implicit Calculus of Constructions [8]. An implicit product $\forall x : \phi.\phi'$ is the type for functions whose arguments are erased during conversion (cf. [9, 2]). Such a type can also be viewed as an infinite intersection type, since its typing rule will assert $\Gamma \vdash t : \forall x : \phi.\phi'$ whenever $\Gamma, x : \phi \vdash t : \phi'$. This rule formalizes (approximately) the idea that t is in the type $\forall x : \phi.\phi'$ whenever it is in each instance of that type (i.e., each type $[u/x]\phi'$ for $u : \phi$). Thus, membership in the $\forall$-type follows from membership in the instances of the body of the $\forall$-type, making the $\forall$-type an intersection of those instances. Note that this is an infinitary intersection, and thus different from the classical finitary intersection type of [4]. We note in passing that the current work includes first-class datatypes, while the other works just cited all rely on encodings of inductive data as lambda terms.

We seek to take the previous approaches further, and erase not just arguments to functions typed with implicit products, but all annotations. This is not the case in the Implicit Calculus of Constructions, for example, or its algorithmic development *ICC** [2], where typing annotations other than implicit arguments are not erased from terms. When testing β-equivalence of terms, we will work with unannotated versions of those terms, where all type- and proof-annotations have been dropped. For associativity of *append*, the proof does not require associativity of *plus*. From the point of view of external reasoning, *append* on vectors will be indistinguishable from *append* on lists (without statically tracked length).

The $\mathtt{T}^{\mathtt{vec}}$ Type Theory. This paper studies versions of a type theory we call $\mathtt{T}^{\mathtt{vec}}$. This system is like Gödel's System T, with vectors and explicit equality proofs. We first study an undecidable version of $\mathtt{T}^{\mathtt{vec}}$ with equality reflection, where terms are completely unannotated (Section 2). We establish standard meta-theoretic results for this unannotated system (Section 3). We then devise a decidable annotated version of the language, which we also call $\mathtt{T}^{\mathtt{vec}}$ (the context will determine whether the annotated or unannotated language is intended). The soundness of annotated $\mathtt{T}^{\mathtt{vec}}$ is justified by erasure to the unannotated system (Section 4). We consider the associativity of *append* in annotated $\mathtt{T}^{\mathtt{vec}}$, as an example (Section 4.1). This approach of studying unannotated versus annotated versions of the type theory should be contrasted with the approach taken in NuPRL, based on Martin-Löf's extensional type theory [3, 6]. There, one constructs typing derivations, as separate artifacts, for unannotated terms. Here, we unite the typing derivation and the unannotated term in a single artifact, namely the annotated term.

Large eliminations. Type-level computation poses challenges for our approach. Because coercions by equality proofs are erased from terms, if we naively extended the system with large eliminations (types defined by pattern matching on terms) we would be able to assign types to diverging or stuck terms. We propose a solution based on what we call *quasi-implicit products*. These effectively serve to mark the introduction and elimination of the intersection type, and prohibit call-by-value reduction within an introduction. This saves Normalization and Progress, which would otherwise fail. We develop the meta-theory of an extension of the unannotated system with large eliminations and call-by-value reduction, including normalization (Section 5).

The basic idea of basing provable equality on the operational semantics of unannotated terms has been implemented previously in the GURU dependently programming language, publicly available at `http://www.guru-lang.org` [10]. The current paper improves upon the work on GURU, by developing and analyzing a formal theory embodying that idea (lacking in [10]).

2 Unannotated $\mathtt{T}^{\mathtt{vec}}$

The definition of unannotated $\mathtt{T}^{\mathtt{vec}}$ uses unannotated terms a (we sometimes also write b):

$$a \ ::= \ x \mid (a\ a') \mid \lambda x.a \mid 0 \mid (S\ a) \mid (R_{\mathtt{nat}}\ a\ a'\ a'') \mid \mathtt{nil} \mid (\mathtt{cons}\ a\ a') \mid (R_{\mathtt{vec}}\ a\ a'\ a'') \mid \mathtt{join}$$

Here, x is for λ-bound variables and S is for successor (not the S combinator). $R_{\mathtt{nat}}$ is the recursor over natural numbers, and $R_{\mathtt{vec}}$ is the recursor over vectors. We have constructors `nil` and `cons` for vectors. The term construct `join` is the introduction form for equality proofs. We will not need an

$$
\begin{array}{lll}
(\lambda x.a)\ a' & \rightsquigarrow & [a'/x]a \\
(R_{\texttt{nat}}\ a\ a'\ 0) & \rightsquigarrow & a \\
(R_{\texttt{nat}}\ a\ a'\ (S\ a'')) & \rightsquigarrow & (a'\ a''\ (R_{\texttt{nat}}\ a\ a'\ a'')) \\
(R_{\texttt{vec}}\ a\ a'\ \texttt{nil}) & \rightsquigarrow & a \\
(R_{\texttt{vec}}\ a\ a'\ (\texttt{cons}\ a_1\ a'')) & \rightsquigarrow & (a'\ a_1\ a''\ (R_{\texttt{vec}}\ a\ a'\ a''))
\end{array}
$$

Figure 1: Reduction semantics for unannotated $\texttt{T}^{\texttt{vec}}$ terms

elimination form, since our system includes a form of equality reflection. For readability, we sometimes use meta-variable l for terms a intended as lengths of vectors. Types ϕ are defined by:

$$\phi \ ::= \ \texttt{nat} \mid \langle \texttt{vec}\ \phi\ a\rangle \mid \Pi x : \phi.\phi' \mid \forall x : \phi.\phi' \mid a = a'$$

The first Π-type is as usual, while the second is an intersection type abstracting a specificational x. This x need not be λ-abstracted in the corresponding term, nor supplied as an argument when that term is applied, similarly to Miquel's implicit products [8].

The reduction relation is the compatible closure under arbitrary contexts of the rules in Figure 1. Figure 2 gives type assignment rules for $\texttt{T}^{\texttt{vec}}$, using a standard definition of typing contexts Γ. We define $\Gamma\ Ok$ to mean that if $\Gamma \equiv \Gamma_1, x : \phi, \Gamma_2$, then $FV(\phi) \subset dom(\Gamma_1)$. We use $a \downarrow a'$ to mean that a and a' are joinable with respect to our reduction relation (i.e., there exists $\hat{a}$ such that $a \rightsquigarrow^* \hat{a}$ and $a' \rightsquigarrow^* \hat{a}$).

Perhaps surprisingly we do not track well-formedness of types, and indeed the `join` and `conv` rules can introduce untypable terms into types. However, they preserve the invariant that terms deemed equal are joinable, and that turns out to be enough to ensure type safety.

Type assignment is not syntax-directed, due to the `(conv)`, `(spec-abs)`, and `(spec-app)` rules, and not obviously decidable. This will not pose a problem here as we study the meta-theoretic properties of the system. Section 4 defines a system of annotated terms which is obviously decidable, and justifies it by translation to unannotated $\texttt{T}^{\texttt{vec}}$. We work up to syntactic identity modulo safe renaming of bound variables, which we denote $\equiv$.

3 Metatheory of Unannotated $\texttt{T}^{\texttt{vec}}$

$\texttt{T}^{\texttt{vec}}$ enjoys standard properties: Type Preservation, Progress (for closed terms), and Strong Normalization. These are all easily obtained, the last by dependency-erasing translation to another type theory (as done originally for LF in [5]). Here, we consider a more semantically informative approach to Strong Normalization. Omitted proofs may be found in a companion report on the second author's web page (see `http://www.cs.uiowa.edu/~astump/papers/ITRS10-long.pdf`).

Theorem 1 (Type Preservation). *If* $\Gamma \vdash a : \phi$ *and* $a \rightsquigarrow a'$*, then* $\Gamma \vdash a' : \phi$.

Theorem 2 (Progress). *If* $\Gamma \vdash a : \phi$ *and* $dom(\Gamma) \cap FV(a) = \emptyset$*, then either* a *is a value or* $\exists a'.a \rightsquigarrow a'$. *Here* a value *is a term of the form*

$$v \ ::= \ \lambda x.a \mid 0 \mid (S\ v) \mid \mathit{nil} \mid (\mathit{cons}\ v\ v') \mid \mathit{join}$$

3.1 Semantics of equality

For our Strong Normalization proof, a central issue is providing an interpretation for equality types in the presence of free variables. We would like to interpret equations like $(\mathit{plus}\ 2\ 2) = 4$ (where the numerals abbreviate terms formed with S and 0 as usual, and plus has a standard recursive definition), as simply

$$\frac{\Gamma(x)\equiv\phi \quad \Gamma\, Ok}{\Gamma\vdash x:\phi}\ \texttt{var}$$

$$\frac{a\downarrow a' \quad \Gamma\, Ok}{\Gamma\vdash \texttt{join} : a=a'}\ \texttt{join} \qquad \frac{\Gamma\vdash a''' : a'=a'' \quad \Gamma\vdash a:[a'/x]\phi \quad x\notin dom(\Gamma)}{\Gamma\vdash a:[a''/x]\phi}\ \texttt{conv}$$

$$\frac{\Gamma, x:\phi'\vdash a:\phi \quad x\notin FV(a)}{\Gamma\vdash a:\forall x:\phi'.\phi}\ \texttt{spec-abs} \qquad \frac{\Gamma\vdash a:\forall x:\phi'.\phi \quad \Gamma\vdash a':\phi'}{\Gamma\vdash a:[a'/x]\phi}\ \texttt{spec-app}$$

$$\frac{\Gamma, x:\phi'\vdash a:\phi}{\Gamma\vdash \lambda x.a:\Pi x:\phi'.\phi}\ \texttt{abs} \qquad \frac{\Gamma\vdash a:\Pi x:\phi'.\phi \quad \Gamma\vdash a':\phi'}{\Gamma\vdash (a\ a'):[a'/x]\phi}\ \texttt{app}$$

$$\frac{\Gamma\, Ok}{\Gamma\vdash 0:\texttt{nat}}\ \texttt{zero} \qquad \frac{\Gamma\, Ok}{\Gamma\vdash \texttt{nil}:\langle\texttt{vec}\ \phi\ 0\rangle}\ \texttt{nil}$$

$$\frac{\Gamma\vdash a:\texttt{nat}}{\Gamma\vdash (S\ a):\texttt{nat}}\ \texttt{succ} \qquad \frac{\begin{array}{l} x\notin dom(\Gamma)\\ \Gamma\vdash a'':\texttt{nat}\\ \Gamma\vdash a:[0/x]\phi\\ \Gamma\vdash a':\Pi y:\texttt{nat}.\Pi u:[y/x]\phi.[(Sy)/x]\phi \end{array}}{\Gamma\vdash (R_{\texttt{nat}}\ a\ a'\ a''):[a''/x]\phi}\ \texttt{Rnat}$$

$$\frac{\begin{array}{l} \Gamma\vdash a:\phi\\ \Gamma\vdash a':\langle\texttt{vec}\ \phi\ l\rangle \end{array}}{\Gamma\vdash (\texttt{cons}\ a\ a'):\langle\texttt{vec}\ \phi\ (S\ l)\rangle}\ \texttt{cons} \qquad \frac{\begin{array}{l} x\notin dom(\Gamma)\\ \Gamma\vdash a'':\langle\texttt{vec}\ \phi'\ l\rangle\\ \Gamma\vdash a:[0/y,\texttt{nil}/x]\phi\\ \Gamma\vdash a':\Pi z:\phi'.\forall l:\texttt{nat}.\Pi v:\langle\texttt{vec}\ \phi'\ l\rangle.\Pi u:[l/y,v/x]\phi.\\ \qquad [(S\ l)/y,(\texttt{cons}\ z\ v)/x]\phi \end{array}}{\Gamma\vdash (R_{\texttt{vec}}\ a\ a'\ a''):[l/y,a''/x]\phi}\ \texttt{Rvec}$$

Figure 2: Type assignment system for unannotated $\texttt{T}^{\texttt{vec}}$

$(plus\ 2\ 2)\downarrow 4$. But when the two terms contain free variables – e.g., in $(plus\ x\ y)=(plus\ y\ x)$ – or when the context is inconsistent, the semantics should make the equation true, even though its sides are not joinable. So our semantics for equality types is joinability under all *ground instances* of the context Γ. The notation for this is $a\sim_\Gamma a'$. The definition must be given as part of the definition of the interpretation of types, because we want to stipulate that the substitutions σ replace each variable x by a ground term in the interpretation of $\sigma\Gamma(x)$. When Γ is empty, we will write $a\sim_\Gamma a'$ as $a\sim a'$. We use a similar convention for other notations subscripted by a context below.

3.2 The interpretation of types

The interpretation of types is given in Figure 3. In that figure, we write $\Rightarrow$ and $\Leftrightarrow$ for meta-level implication and equivalence, respectively, and give $\Leftrightarrow$ lowest precedence among all infix symbols, and $\Rightarrow$ next lowest precedence. We stipulate up front (not in the clauses in the figure) that $a\in[\![\phi]\!]_\Gamma$ requires $a\in SN$ (where SN is the set of strongly normalizing terms) and $\Gamma\vdash a:\phi$. The definition in Figure 3 proceeds by well-founded recursion on the triple $(|\Gamma|, d(\phi), l(a))$, in the natural lexicographic ordering. Here, $|\Gamma|$ is the cardinality of $dom(\Gamma)$, and if $a\in SN$, then we make use of a (finite) natural number $l(a)$ bounding

$$\begin{array}{lcl} a \in [\![\mathtt{nat}]\!]_\Gamma & \Leftrightarrow & \top \\ a \in [\![\langle \mathtt{vec}\ \phi\ l\rangle]\!]_\Gamma & \Leftrightarrow & (a \rightsquigarrow^* \mathtt{nil} \ \Rightarrow\ l \sim_\Gamma 0) \wedge \\ & & \forall a'.\forall a''. a \rightsquigarrow^* (\mathtt{cons}\ a'\ a'') \ \Rightarrow\ \begin{array}[t]{l} (i)\ a' \in [\![\phi]\!]_\Gamma \ \wedge\ \exists l'. \\ (ii)\ a'' \in [\![\langle \mathtt{vec}\ \phi\ l'\rangle]\!]_\Gamma \wedge \\ (iii)\ l \sim_\Gamma (S\ l') \end{array} \\ a \in [\![\Pi x : \phi'.\phi]\!]_\Gamma & \Leftrightarrow & \forall a' \in [\![\phi']\!]^+_\Gamma.\ (a\ a') \in [\![[a'/x]\phi]\!]_\Gamma \\ a \in [\![\forall x : \phi'.\phi]\!]_\Gamma & \Leftrightarrow & \forall a' \in [\![\phi']\!]^+_\Gamma.\ a \in [\![[a'/x]\phi]\!]_\Gamma \\ a \in [\![a_1 = a_2]\!]_\Gamma & \Leftrightarrow & (a \rightsquigarrow^* \mathtt{join} \Rightarrow a_1 \sim_\Gamma a_2) \end{array}$$

<u>where</u>:

$$\begin{array}{lcl} a \sim_\Gamma a' & \Leftrightarrow & \forall \sigma.\ \sigma \in [\![\Gamma]\!] \ \Rightarrow\ (\sigma a) \downarrow (\sigma a') \\ a \in [\![\phi]\!]^+_\Gamma & \Leftrightarrow & a \in [\![\phi]\!]_\Gamma \ \wedge\ (|\Gamma| > 0 \ \Rightarrow\ \forall \sigma \in [\![\Gamma]\!].\ \sigma a \in [\![\sigma\phi]\!]) \end{array}$$

<u>and also</u>:

$$\frac{}{\emptyset \in [\![\cdot]\!]_\Delta} \qquad \frac{a \in [\![\sigma\phi]\!]^+_\Delta \quad \sigma \in [\![\Gamma]\!]_\Delta}{\sigma \cup \{(x,a)\} \in [\![\Gamma, x : \phi]\!]_\Delta}$$

Figure 3: The interpretation $a \in [\![\phi]\!]_\Gamma$ of strongly normalizing terms with $\Gamma \vdash a : \phi$

the number of symbols in the normal form of a. We need to assume confluence of reduction elsewhere in this proof, so it does not weaken the result to assume here that each term has at most one normal form. While we believe confluence for this language should be easily established by standard methods, that proof remains to future work. The quantity $d(\phi)$ is the depth of ϕ, defined as follows:

$$\begin{array}{lcl@{\qquad}lcl} d(\mathtt{nat}) & = & 0 & d(\langle \mathtt{vec}\ \phi\ l\rangle) & = & 1 + d(\phi) \\ d(\Pi x : \phi.\phi') & = & 1 + max(d(\phi), d(\phi')) & d(\forall x : \phi.\phi') & = & 1 + max(d(\phi), d(\phi')) \\ d(a = a') & = & 0 \end{array}$$

Note that $d(\phi) = d([a/x]\phi)$ for all a, x, and ϕ. Also, in the clause for vec-types, since the right hand side of the clause conjoins the condition $a \in SN$, $l(a)$ is defined, and we have $l(a'') < l(\mathtt{cons}\ a'\ a'')$. The figure gives an inductive definition for when $\sigma \in [\![\Gamma]\!]_\Delta$. We call such a σ a *closable substitution*.

In general, the inductive definition of closable substitution $\sigma \in [\![\Gamma]\!]_\Delta$ allows the range of the substitution to contain open terms. When Δ is empty, σ is a *closing* substitution. The definition of $[\![\cdot]\!]$ for types uses the definition of closable substitutions in a well-founded way. We appeal only to $[\![\Gamma]\!]$ (with an empty context Δ) in the definitions of $[\![\phi]\!]_\Gamma$ and $[\![\phi]\!]^+_\Gamma$. Where the definition of $[\![\Gamma]\!]_\Delta$ appeals back to the interpretation of types, it does so only when this Γ was non-empty, and with an empty context given for the interpretation of the type. So $|\Gamma|$ has indeed decreased from one appeal to the interpretation of types to the next.

3.3 Critical properties

A term is defined to be *neutral* iff it is of the form $(a\ a')$ or $(R_B\ a\ a'\ a'')$ (with $B \in \{\mathtt{nat}, \mathtt{vec}\}$), or if it is a variable. We prove three critical properties of reducibility at type ϕ, by mutual induction on $(|\Gamma|, d(\phi), l(a))$. Here we write $next(a) = \{a' \mid a \rightsquigarrow a'\}$.

R-Pres. If $a \in [\![\phi]\!]_\Gamma$, then $next(a) \subset [\![\phi]\!]_\Gamma$.
R-Prog. If a is neutral and $\Gamma \vdash a : \phi$, then $next(a) \subset [\![\phi]\!]_\Gamma$ implies $a \in [\![\phi]\!]_\Gamma$.
R-Join. Suppose $a_1 \sim_\Gamma a_2$; $\Gamma \vdash a' : a_1 = a_2$ for some a'; and $x \notin dom(\Gamma)$. Then $[\![[a_1/x]\phi]\!]_\Gamma \subset [\![[a_2/x]\phi]\!]_\Gamma$.

3.4 Soundness of typing with respect to the interpretation

Our typing rules are sound with respect to our interpretation of types (Figure 3). As usual, we must strengthen the statement of soundness for the induction to go through. We need a quasi-order $\subset$ on contexts, defined by: $\Delta \subset \Gamma \Leftrightarrow \forall x \in dom(\Delta).\ \Delta(x) = \Gamma(x)$.

Theorem 3 (Soundness for Interpretations)**.** *Suppose $\Gamma \vdash a : \phi$. Then for any ΔOk with $\Delta \subset \Gamma$ and $\sigma \in [\![\Gamma]\!]_\Delta$, we have $(\sigma a) \in [\![\sigma\phi]\!]_\Delta$.*

Critically, we quantify over possibly open substitutions σ, whose ranges consist of closable terms.

Corollary 1 (Strong Normalization)**.** *If $\Gamma \vdash a : \phi$, then $a \in SN$.*

Corollary 2. *If $\Gamma \vdash a : \phi$ and $\Gamma \vdash a' : \phi'$, then $a \downarrow a'$ is decidable.*

Corollary 3 (Equational Soundness)**.** *If $\cdot \vdash a : b_1 = b_2$, then $b_1 \downarrow b_2$.*

Corollary 4 (Logical Soundness)**.** *There is a type ϕ such that $\vdash a : \phi$ does not hold for any a.*

Proof. By Equational Soundness, we do not have $\vdash a : 0 = (S\ 0)$ for any a.

4 Annotated $\mathtt{T}^{\mathtt{vec}}$

We now define a system of annotated terms t, and a decidable type computation system deriving judgments $\Gamma \Vdash t : \phi$, justified by dropping annotations via $|\cdot|$ (defined in Figure 4). The annotated terms t are the following. Annotations include types ϕ, possibly with designated free variables, as in $x.\phi$ (bound by the dot notation).

$$\begin{array}{rcl} t & ::= & x \mid (t\ t') \mid (t\ t')^- \mid \lambda x : \phi.t \mid \lambda^- x : \phi.t \mid 0 \mid (S\ t) \mid (R_{\mathtt{nat}}\ x.\phi\ t\ t'\ t'') \\ & & \mid (\mathtt{nil}\ \phi) \mid (\mathtt{cons}\ t\ t') \mid (R_{\mathtt{vec}}\ x.y.\phi\ t\ t'\ t'') \mid (\mathtt{join}\ t\ t') \mid (\mathtt{cast}\ x.\phi\ t\ t') \end{array}$$

Three new constructs correspond to the typing rules `(spec-abs)`, `(spec-app)`, and `(conv)` of Figure 2: $\lambda^- x : \phi'.\phi$, $(t\ t')^-$ and $(\mathtt{cast}\ x.\phi\ t\ t')$. Figure 5 gives syntax-directed type-computation rules, which constitute a deterministic algorithm for computing a type ϕ as output from a context Γ and annotated term t as inputs. Several rules use the $|\cdot|$ function, since types ϕ (as defined in Section 2 above) may mention only unannotated terms.

Theorem 4 (Algorithmic Typing)**.** *Given Γ and a, we can, in an effective way, either find ϕ such that $\Gamma \Vdash a : \phi$, or else report that there is no such ϕ.*

This follows in a standard way from inspection of the rules, using Corollary 2 for the `join`-rule.

Theorem 5 (Soundness for Type Assignment)**.** *If $\Gamma \Vdash t : \phi$ then $\Gamma \vdash |t| : \phi$.*

4.1 Example

Now let us see versions of the examples mentioned in Section 1, available in the `guru-lang/lib/vec.g` library file for GURU (see `www.guru-lang.org`). The desired types for vector append ("*append*") and for associativity of vector append are:

$$\begin{array}{lcl} \mathit{append} & : & \forall l_1 : \mathtt{nat}.\forall l_2 : \mathtt{nat}.\Pi v_1 : \langle \mathtt{vec}\ \phi\ l_1\rangle.\Pi v_2 : \langle \mathtt{vec}\ \phi\ l_2\rangle.\langle \mathtt{vec}\ \phi\ (\mathit{plus}\ l_1\ l_2)\rangle \\ \mathit{append\_assoc} & : & \forall l_1 : \mathtt{nat}.\forall l_2 : \mathtt{nat}.\forall l_3 : \mathtt{nat}. \\ & & \Pi v_1 : \langle \mathtt{vec}\ \phi\ l_1\rangle.\Pi v_2 : \langle \mathtt{vec}\ \phi\ l_2\rangle.\Pi v_3 : \langle \mathtt{vec}\ \phi\ l_3\rangle. \\ & & \quad (\mathit{append}\ (\mathit{append}\ v_1\ v_2)\ v_3) = (\mathit{append}\ v_1\ (\mathit{append}\ v_2\ v_3)) \end{array}$$

$$\begin{array}{lcllcl}
|x| & = & x & |(t\ t')| & = & (|t|\ |t'|) \\
|(t\ t')^-| & = & |t| & |\lambda x:\phi.t| & = & \lambda x.|t| \\
|\lambda^- x:\phi.t| & = & |t| & |0| & = & 0 \\
|(S\ t)| & = & (S\ |t|) & |(\texttt{nil}\ \phi)| & = & \texttt{nil} \\
|(\texttt{cons}\ t\ t')| & = & (\texttt{cons}\ |t|\ |t'|) & |(R_{\texttt{nat}}\ x.\phi\ t\ t'\ t'')| & = & (R_{\texttt{nat}}\ |t|\ |t'|\ |t''|) \\
|(R_{\texttt{vec}}\ x.y.\phi\ t\ t'\ t'')| & = & (R_{\texttt{vec}}\ |t|\ |t'|\ |t''|) & |(\texttt{join}\ t\ t')| & = & \texttt{join} \\
|(\texttt{cast}\ x.\phi\ t\ t')| & = & |t'| & & &
\end{array}$$

Figure 4: Translation from annotated terms to unannotated terms

$$\frac{\Gamma \Vdash t:\phi \quad \Gamma \Vdash t':\phi' \quad |t| \downarrow |t'|}{\Gamma \Vdash (\texttt{join}\ t\ t') : |t| = |t'|} \qquad \frac{\Gamma \Vdash t : a = a' \quad \Gamma \Vdash t' : [a/x]\phi}{\Gamma \Vdash (\texttt{cast}\ x.\phi\ t\ t') : [a'/x]\phi} \qquad \frac{\Gamma, x:\phi' \Vdash t:\phi \quad x \notin FV(|t|)}{\Gamma \Vdash \lambda^- x:\phi'.t : \forall x:\phi'.\phi}$$

$$\frac{\Gamma \Vdash t : \forall x:\phi'.\phi \quad \Gamma \Vdash t':\phi'}{\Gamma \Vdash (t\ t')^- : [|t'|/x]\phi} \qquad \frac{\Gamma, x:\phi' \Vdash t:\phi}{\Gamma \Vdash \lambda x:\phi'.t : \Pi x:\phi'.\phi} \qquad \frac{\Gamma \Vdash t : \Pi x:\phi'.\phi \quad \Gamma \Vdash t':\phi'}{\Gamma \Vdash (t\ t') : [|t'|/x]\phi}$$

$$\frac{\begin{array}{l}\Gamma \Vdash t'' : \langle \texttt{vec}\ \phi'\ l\rangle \\ \Gamma \Vdash t : [0/x, \texttt{nil}/y]\phi \\ \Gamma \Vdash t' : \forall l:\texttt{nat}.\Pi z:\phi'.\Pi v:\langle \texttt{vec}\ \phi'\ l\rangle.\Pi u:[l/x,v/y]\phi. \\ \qquad [(S\ l)/x, (\texttt{cons}\ z\ v)/y]\phi\end{array}}{\Gamma \Vdash (R_{\texttt{vec}}\ x.y.\phi\ t\ t'\ t'') : [l/x, |t''|/y]\phi}$$

Figure 5: Type-computation system for annotated $\texttt{T}^{\texttt{vec}}$ (selected rules)

We consider now annotated inhabitants of these types. The first is the following:

$$\begin{array}{lcl}
\mathit{append} & = & \lambda^- l_1:\texttt{nat}.\lambda^- l_2:\texttt{nat}.\lambda v_1:\langle \texttt{vec}\ \phi\ l_1\rangle.\lambda v_2:\langle \texttt{vec}\ \phi\ l_2\rangle. \\
& & \quad (R_{\texttt{vec}}\ (x.y.\langle \texttt{vec}\ \phi\ (\mathit{plus}\ x\ l_2)\rangle) \\
& & \qquad (\texttt{cast}\ (x.\langle \texttt{vec}\ \phi\ x\rangle)\ P_1\ v_2) \\
& & \qquad (\lambda^- l:\texttt{nat}.\lambda x:\phi.\lambda v_1':\langle \texttt{vec}\ \phi\ l\rangle.\lambda r:\langle \texttt{vec}\ \phi\ (\mathit{plus}\ l\ l_2)\rangle). \\
& & \qquad\quad (\texttt{cast}\ (x.\langle \texttt{vec}\ \phi\ x\rangle)\ P_2\ (\texttt{cons}\ x\ r)) \\
& & \qquad v_1)
\end{array}$$

The two cases in the $R_{\texttt{vec}}$ term return a type-cast version of what would standardly be returned in an unannotated version of *append*. The proofs P_1 and P_2 used in those casts show respectively that $l_2 = (\mathit{plus}\ 0\ l_2)$ and $(S\ (\mathit{plus}\ l\ l_2)) = (\mathit{plus}\ (S\ l)\ l_2)$. They are simple join-proofs:

$$P_1 \quad = \quad (\texttt{join}\ l_2\ (\mathit{plus}\ 0\ l_2)) \qquad\qquad P_2 \quad = \quad (\texttt{join}\ (S\ (\mathit{plus}\ l\ l_2))\ (\mathit{plus}\ (S\ l)\ l_2))$$

Now for *append_assoc*, we can use the following annotated term:

$$\begin{array}{lcl}
\mathit{append\_assoc} & = & \lambda^- l_1:\texttt{nat}.\lambda^- l_2:\texttt{nat}.\lambda^- l_3:\texttt{nat}. \\
& & \lambda v_1:\langle \texttt{vec}\ \phi\ l_1\rangle.\lambda v_2:\langle \texttt{vec}\ \phi\ l_2\rangle.\lambda v_3:\langle \texttt{vec}\ \phi\ l_3\rangle. \\
& & \quad (R_{\texttt{vec}}\ (x.y.(\mathit{append}\ (\mathit{append}\ v_1\ v_2)\ v_3) = (\mathit{append}\ v_1\ (\mathit{append}\ v_2\ v_3))) \\
& & \qquad (\texttt{join}\ (\mathit{append}\ (\mathit{append}\ \texttt{nil}\ v_2)\ v_3) = (\mathit{append}\ \texttt{nil}\ (\mathit{append}\ v_2\ v_3))) \\
& & \qquad (\lambda^- l:\texttt{nat}.\lambda x:\phi.\lambda v_1':\langle \texttt{vec}\ \phi\ l\rangle. \\
& & \qquad\ \lambda r:(\mathit{append}\ (\mathit{append}\ v_1'\ v_2)\ v_3) = (\mathit{append}\ v_1'\ (\mathit{append}\ v_2\ v_3)). \\
& & \qquad\quad P_3))
\end{array}$$

$$\phi \ ::= \ \ldots \mid \texttt{ifZero}\, a\, \phi\, \phi' \qquad a \ ::= \ \ldots \mid \lambda.a \mid a\,\Box \qquad v \ ::= \ \ldots \mid \lambda.a$$

$$\frac{\Gamma, x:\phi' \vdash a:\phi \quad x \notin FV(a)}{\Gamma \vdash \lambda.a : \forall x:\phi'.\phi}\ \texttt{spec-abs'} \qquad \frac{\Gamma \vdash a : \forall x:\phi'.\phi \quad \Gamma \vdash a' : \phi'}{\Gamma \vdash a\,\Box : [a'/x]\phi}\ \texttt{spec-app'}$$

$$\frac{\Gamma \vdash a:\phi}{\Gamma \vdash a : \texttt{ifZero}\, 0\, \phi\, \phi'}\ \texttt{foldZ} \qquad \frac{\Gamma \vdash a : \texttt{ifZero}\, 0\, \phi\, \phi'}{\Gamma \vdash a:\phi}\ \texttt{unfoldZ}$$

$$\frac{\Gamma \vdash a:\phi' \quad \Gamma \vdash a' : \texttt{nat}}{\Gamma \vdash a : \texttt{ifZero}\, (S\, a')\, \phi\, \phi'}\ \texttt{foldS} \qquad \frac{\Gamma \vdash a : \texttt{ifZero}\, (S\, a')\, \phi\, \phi' \quad \Gamma \vdash a' : \texttt{nat}}{\Gamma \vdash a:\phi'}\ \texttt{unfoldS}$$

Figure 6: Types, terms, values, and typing rules for $\texttt{T}^{\texttt{vec}}$ with large eliminations.

The omitted proof P_3 is an easy equational proof of the following type:

$$(\mathit{append}\ (\mathit{append}\ (\texttt{cons}\ x\ v_1')\ v_2)\ v_3) = (\mathit{append}\ (\texttt{cons}\ x\ v_1')\ (\mathit{append}\ v_2\ v_3))$$

5 $\texttt{T}^{\texttt{vec}}$ with Large Eliminations

Next we study an extended version of $\texttt{T}^{\texttt{vec}}$ with large eliminations, i.e. types defined by pattern matching on terms. This extended language no longer is normalizing under general β-reduction $\rightsquigarrow$, but we will prove that well-typed closed terms normalize under call-by-value evaluation $\rightsquigarrow_v$. In particular, the language is type safe and logically consistent.

The additions to the language and type system are shown in figure 6.

The type language is extended with the simplest possible form of large elimination, a type-level conditional `ifZero`which is introduced and eliminated by the `fold` and `unfold` rules. While type conversion and type folding/unfolding are completely implicit, we replace the `spec-abs/app` rules with new rules `spec-abs'/app'` which require the place where we introduce or eliminate the $\forall$-type to be marked by new *quasi-implicit* forms $\lambda.a$ and $a\ \Box$. These forms do not mention the quantified variable or the term it is instantiated with, so we retain the advantages of specificational reasoning. The point of these forms is their evaluation behavior: $(\lambda.a)\ \Box \rightsquigarrow_v a$, and $\lambda.a$ counts as a value so CBV evaluation will never reduce inside it. Besides this, the CBV operational semantics is standard, so we omit it here.

In the language with large eliminations we no longer have normalization or type safety for arbitrary open terms. This is because the richer type system lets us make use of absurd equalities: whenever we have $\Gamma \vdash a : \phi$ and $\Gamma \vdash p : (S\ a'){=}0$, we can show $\Gamma \vdash a : \phi'$ for any ϕ' by going via the intermediate type $(\texttt{ifZero}\ 0\ \phi(\alpha.\phi'))$. In particular, this means we can show judgments like

$$p : 1{=}0 \vdash (\lambda x.x\,x)\ (\lambda x.x\,x) : \texttt{nat} \qquad \text{and} \qquad p : 1{=}0 \vdash 0\ 0 : \texttt{nat}.$$

This is also the reason we introduce the quasi-implicit products. Using our old rule `spec-abs` we would be able to show $\vdash 0\ 0 : \forall p : 1{=}0.\texttt{nat}$, despite $0\ 0$ being a stuck term in our operational semantics.

Because of this *quod libet* property it is no longer convenient to prove Progress and Preservation before Normalization. While the proof of Preservation is not hard, Progress as we have seen depends on the logical consistency of the language, which is exactly what we hope to establish through Normalization. To cut this circle we design an interpretation of types (figure 7) that lets us prove type safety, Canonical Forms and Normalization in a single induction.

$$
\begin{array}{lcl}
a \in [\![\texttt{nat}]\!] & \Leftrightarrow & \exists n. a \leadsto_v^* n \\
a \in [\![\langle \texttt{vec}\ \phi\ l \rangle]\!] & \Leftrightarrow & (a \leadsto_v^* \texttt{nil} \wedge l \leadsto^* 0) \vee \\
 & & \quad \exists v\ v'\ n.\quad a \leadsto_v^* (\texttt{cons}\ v\ v') \wedge l \leadsto^* (S\ n) \\
 & & \qquad\qquad \wedge\ v \in [\![\phi]\!]\ \wedge\ v' \in [\![\langle \texttt{vec}\ \phi\ n \rangle]\!] \\
a \in [\![\Pi x : \phi'.\phi]\!] & \Leftrightarrow & \exists a'.a \leadsto_v^* (\lambda x.a')\ \wedge\ \forall a' \in [\![\phi']\!].\ (a\ a') \in [\![[a'/x]\phi]\!] \\
a \in [\![\forall x : \phi'.\phi]\!] & \Leftrightarrow & \exists a'.a \leadsto_v^* (\lambda a')\ \wedge\ \forall a' \in [\![\phi']\!].\ (a\ \Box) \in [\![[a'/x]\phi]\!] \\
a \in [\![a_1 = a_2]\!] & \Leftrightarrow & a \leadsto_v^* \texttt{join}\ \wedge\ a_1 \downarrow a_2 \\
a \in [\![\texttt{ifZero}\ b\ \phi\ \phi']\!] & \Leftrightarrow & \begin{cases} a \in [\![\phi]\!] & \text{if } b \leadsto^* 0 \\ a \in [\![\phi']\!] & \text{if } b \leadsto^* (S\ n) \\ \text{False} & \text{otherwise} \end{cases}
\end{array}
$$

$$
\frac{}{\emptyset \in [\![\cdot]\!]} \qquad \frac{v \in [\![\sigma\phi]\!] \quad \sigma \in [\![\Gamma]\!]}{\sigma \cup \{(x,v)\} \in [\![\Gamma, x : \phi]\!]}
$$

Figure 7: Type interpretation $a \in [\![\phi]\!]$ and context interpretation $\sigma \in [\![\Gamma]\!]$ for $\texttt{T}^{\texttt{vec}}$ with large eliminations

5.1 Semantics of Equality

We need to pick an interpretation for equality types. Since we are only interested in closed terms, this can be less elaborate than in section 3. Perhaps surprisingly, even though we are interested in CBV-evaluation of programs, we can still interpret equality as joinability $\downarrow$ under unrestricted β-reduction. In the interpretation we use $\leadsto_v$ for the program being evaluated, but $\leadsto$ whenever we talk about terms occurring in types (namely in `vec`, $=$, and R-types). The `join` typing rule is specified in terms of $\leadsto$, so when doing symbolic evaluation of programs at type checking time the type checker can use unrestricted reduction, which gives a powerful type system than can prove many equalities.

5.2 Normalization to Canonical Form

We define the interpretation $[\![\ ]\!]$ as in figure 7 by recursion on the depth of the type ϕ. As we only deal with closed terms, the definition can be simpler than the one in section 3. The proof then proceeds much like the proof for open terms:
R-Canon. If $a \in [\![\phi]\!]$, then $a \leadsto_v^* v$ for some v. Furthermore, if the top-level constructor of ϕ is `nat`, Π, $\forall$, $=$, or `vec`, then v is the corresponding introduction form.
R-Pres. If $a \in [\![\phi]\!]$ and $a \leadsto_v a'$, then $a' \in [\![\phi]\!]$.
R-Prog. If $a \leadsto_v a'$, and $a' \in [\![\phi]\!]$, then $a \in [\![\phi]\!]$.
R-Join. If $a_1 \downarrow a_2$, then $a \in [\![[a_1/x]\phi]\!]$ implies $a \in [\![[a_2/x]\phi]\!]$.

Theorem 6. *If $\Gamma \vdash a : \phi$ and $\sigma \in [\![\Gamma]\!]$, then $\sigma a \in [\![\sigma\phi]\!]$.*

Corollary 5 (Type Safety). *If $\vdash a : \phi$, then $a \leadsto_v^* v$.*

Corollary 6 (Logical Soundness). *$\vdash a : 1{=}0$ does not hold for any a.*

6 Conclusion and Future Work

The $\texttt{T}^{\texttt{vec}}$ type theory includes intersection types and a form of equality reflection, justified by translation to an undecidable unannotated system. The division into annotated and unannotated systems enables us to reason about terms without annotations, while retaining decidable type checking. We have seen how this approach extends to a language including large eliminations, by introducing a novel kind of *quasi-implicit* products. The quasi-implicit products allow convenient reasoning about specificational data, while permitting a simple proof of normalization of closed terms. Possible future work includes formalizing the metatheory, and extending to a polymorphic type theory. Adding an extensional form of equality while retaining decidability would also be of interest, as in [1].

Acknowledgments: Thanks to members of the TRELLYS team, especially Stephanie Weirich and Tim Sheard, for discussions on this and related systems. This work was partially supported by the the U.S. National Science Foundation under grants 0910510 and 0910786.

References

[1] T. Altenkirch, C. McBride, and W. Swierstra. Observational Equality, Now! In A. Stump and H. Xi, editors, *PLPV '07: Proceedings of the 2007 Workshop on Programming Languages meets Program Verification*, pages 57–68, 2007.

[2] 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.

[3] R. Constable and the PRL group. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.

[4] M. Coppo1 and M. Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv. Math. Logik*, 19(1):139–156, 1978.

[5] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[6] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

[7] C. McBride. *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, University of Edinburgh, 1999.

[8] A. Miquel. The Implicit Calculus of Constructions. In *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 344–359. Springer, 2001.

[9] 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)*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2008.

[10] 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)*, pages 49–58, 2009.