Verified Programming in Guru

Aaron Stump
CS Dept.
The University of Iowa, USA
astump@acm.org

Morgan Deters
LSI Dept.
Universitat Politècnica de Catalunya,
Spain
mdeters@lsi.upc.edu

Adam Petcher , Todd Schiller
Timothy Simpson
CSE Dept.
Washington University in St. Louis, USA
adampetcher@yahoo.com,
{tws2,tas5}@cec.wustl.edu

Abstract

Operational Type Theory (OpTT) is a type theory allowing possibly diverging programs while retaining decidability of type checking and a consistent logic. This is done by distinguishing proofs and (program) terms, as well as formulas and types. The theory features propositional equality on type-free terms, which facilitates reasoning about dependently typed programs. OpTT has been implemented in the GURU verified programming language, which includes a type- and proof-checker, and a compiler to efficient C code. In addition to the core OpTT, GURU implements a number of extensions, including ones for verification of programs using mutable state and input/output. This paper gives an introduction to verified programming in GURU.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Applicative (functional) languages; F.3.1 [*Logics and Meanings of Programs*]: Mechanical verification; F.4.1 [*Mathematical Logic*]: Mechanical theorem proving

General Terms Languages, Verification

Keywords Operational Type Theory, Dependently Typed Programming, Language-Based Verification

1. Introduction

Dependently typed programming languages are a subject of considerable recent attention from the Programming Languages and more traditional Type Theory communities. The goal of this paper is to describe verified programming practice in one such language, called GURU. GURU is a verified programming language, which combines a dependently typed, pure functional programming language with a sound logical theory of its untyped evaluation. Using this theory, properties may be proved about program terms with all their annotations dropped. Such annotations include types, proofs used in explicit casts, and specificational data. Explicit casts are used to change the type checker's view of a term beyond what the range of GURU's (quite weak) definitional equality. For example, in GURU, the type of vectors of A's of length $3 + 3$, written $\langle \texttt{vec}\ A\ (\texttt{plus}\ 3\ 3)\rangle$, is not definitionally equal to $\langle \texttt{vec}\ A\ 6\rangle$. The presence of applications of possibly diverging functions in types, which is allowed in GURU, generally precludes making two such types definitionally equal. They are, however, provably equal, and a cast with such a proof can be used to change a term of the former type into one with the latter. Casts have no computational effect, and are removed both during compilation and when dropping annotations for provable equality. Annotations are dropped by definitional equality, which greatly reduces the burden on the programmer to ensure that such annotations match up at various points in code and proofs. An example of specificational data could be the length of such a vector. Some programs never do a case analysis of the length of a vector, but only the vector itself, using the length just for specificational purposes. In GURU, inputs to functions, including term constructors, may be designated as specificational. A simple analysis checks that no specificational data is ever used in a computational position. The data may then safely be dropped during compilation and theorem proving.

Having definitional equality drop annotations turns out to be a remarkably robust design idea for extensions. We discuss several such extensions in this paper. One is termination casts. These are like type casts but change the type checker's view of the termination behavior of a term. In particular, casting an application of terminating terms with a termination cast results in a terminating term. Certain positions in code and proofs are restricted to terminating terms only. These include instantiations of quantifiers, which range over values of the quantified type, excluding non-terminating terms; and also specificational arguments to functions, which should be required to be terminating since those arguments will not be executed when the compiled program is run.

Another extension is explicit increments and decrements of reference counts for data. In GURU, datatypes are, by default, reference counted. Explicit increments and decrements are included in code to manipulate reference counts. Memory safety and lack of memory leaks is guaranteed by a static analysis that enforces a data ownership policy on code, which ensures that the reference count is greater than zero whenever a decrement occurs, and that all data eventually have their reference counts decremented to zero. By default, function arguments are owned by the callee, and must be consumed (for example by decrementing the reference count) by the time it returns. Arguments may, however, be designated as owned by the caller, in which case, it is not necessary for the function to consume them. This approach helps reduce the number of increments and decrements. Together with size-segregated free lists, this leads to good performance results on a number of benchmarks. While increments and decrements are critically relevant to these memory properties of compiled code, they are otherwise irrelevant to its behavior as specified by its operational semantics, which, as standard, does not address memory allocation issues. GURU's logical theory is concerned only with this operational behavior, and hence for theorem proving purposes, increments and decrements

PLPV'09, January 20, 2009, Savannah, Georgia, USA.

are computationally irrelevant and can be dropped like other annotations by definitional equality.

GURU's ownership analysis naturally accommodates an additional extension, namely linear types. As proposed in previous work, linear types may be used for verified practical programming to give programmers tight control over resource usage (Zhu and Xi 2005). In GURU, this control is put to work for functional modeling. Following an idea of Swierstra and Altenkirch, GURU supports non-functional constructs by describing them with a pure functional model (Swierstra and Altenkirch 2007). For example, mutable arrays of length N can be modeled as vectors of length N. The critical operations of reading and writing an array are done using the obvious linear time operations in the model. During compilation, however, the functional model (with its critical operations) is replaced by the more efficient non-functional model. For this approach to be sound, it is necessary to restrict usage of the functional model in such a way that the behavior mandated by the operational semantics will be consistent with that resulting from the non-functional model in the compiled code. Swierstra and Altenkirch propose to restrict their usage with monads. In GURU, their usage is restricted with linear types.

The rest of the paper covers the above surveyed ideas in more detail, with a focus on using the described features in practice. A theoretical treatment of the core OPTT is described elsewhere (Stump and Westbrook). A theoretical treatment of the other features is left to future work. We begin with quick consideration of related verified programming implementations. Then we will briefly consider the syntax and classification rules for terms and types. Proofs are distinguished in GURU from terms, and formulas from types, again for purposes of soundly accommodating possibly diverging terms. The classification rules for proofs and formulas are mostly omitted here. The interested reader can find them in (Stump and Westbrook).

2. Related Work

EPIGRAM is a total type theory proposed for practical programming with dependent types (McBride and McKinna 2004). Agda has similar aims (Norell 2007). Xi's ATS and a system proposed by Licata and Harper have similar aims, but allow general recursion (Licata and Harper 2005; Chen and Xi 2005). Programs which can index types and which are subject to external reasoning, however, are required to be uniformly terminating. This is done via *stratification*: such terms are drawn from a syntactic class distinct from that of program terms. Existing stratified systems restrict external verification to terms in the index domain. Similar approaches are taken in CONCOQTION and ΩMEGA (Pasalic et al. 2007; Sheard 2006). In contrast, OPTT supports what is coming to be called "full-spectrum dependency", where arbitrary program terms may appear in types. Hoare Type Theory supports internal verification of possibly non-terminating, imperative programs, with a relatively low-level model of the heap (Nanevski and Morrisett 2005). In contrast, OPTT supports internal and external verification of pure functional programs, with mutable state accommodated using linear types. OPTT's approach sacrifices a low-level view of the heap in favor, it is hoped, for a more manageable level of abstraction.

3. Central Design Ideas of OPTT

The central design idea of OPTT is to separate three uses of reduction, to apply different restrictions to each: *definitional* reduction in definitional equality, which in OPTT is restricted to be very weak, only dropping annotations, α-equivalence, and unfolding non-recursive definitions; *computational* reduction of program terms for purposes of computing a value, which is not restricted at all; and *logical* reduction of proofs to establish logical consistency, which in OPTT is restricted to be of relatively modest complexity, disallowing higher-order logical features. The advantage of this approach is that we may retain a relatively simple logic, in a proof theoretic sense, for reasoning about possibly non-terminating programs. Decidability of type checking requires that reduction for definitional equality be strictly weaker than reduction for computing a value. There are other reasons for keeping reduction for definitional equality weak, which are discussed in (Stump and Westbrook). The rest of the language design unfolds from this starting point. Because definitional equality is weak, we know we will need casts in terms, which motivates OPTT's untyped provable equality. It is otherwise well known to be tedious to reason about dependently typed programs, due to the need to deal with casts in terms. The mechanism used for separating computational reduction and logical reduction is simply to have separate syntax for proofs and program terms, and hence for formulas and types.

4. Term and Type Syntax

The syntax for OPTT terms and types is given in Figure 1. The syntax-directed classification rules for terms are given in Figure 2. All classification rules in this paper compute a classifier as output for a given context Γ and expression as input. Contexts assign classifiers to constants and variables. Declarations for constants are assumed added, as discussed in Section 4.4 below. The rules operate modulo definitional equality. Their syntax-directed nature implies decidability of classification.

Meta-variables. We write P for proofs, and F for formulas, defined in Section 5. We use x for variables, c for term constructors, and d for type constructors. We occasionally use v for any variable or term constructor. Variables are considered to be syntactically distinguished as either term-, type-, or proof-level. This enables definitional equality to recognize which variables are proofs or types. A reserved constant `!` is used for erased annotations, including types (Section 4.1).

Specificationality. We write o for ownership annotations on function inputs. These will be extended below, but for now, the only such is `spec`, for specificationality. Similarly, s is for indicating specificationality of arguments in applications: either `spec` or nothing. The reason for marking specificationality in applications syntactically is to keep definitional equality from depending on typing. A simple static *specificationality analysis* ensures that specificational inputs are used only as specificational arguments to functions. They are disallowed everywhere else. This specificationality analysis is performed separately from typing, and so the typing rules ignore specificationality annotations. Currently in GURU, the programmer explicitly marks function inputs as specificational, while the specificationality annotations for applications are inferred during type checking. Occasionally the programmer finds it useful to supply specificationality annotations on applications him/herself, which GURU allows.

Multi-arity notations. We write $\texttt{fun}\ x(\bar{o}\ \bar{x}\ :\ \bar{A})\ :\ T.\ t$ for $\texttt{fun}\ x(o_1\ x_1\ :\ A_1)\cdots(o_n\ x_n\ :\ A_n)\ :\ T.\ t$, with $n > 0$, and each o_i either `spec` or nothing. Also, in the `fun` typing rule, we use judgment $\Gamma \vdash \bar{x} : \bar{A}$:

$$\frac{\Gamma \vdash A : \mathit{sort}(A) \quad \Gamma, x : A \vdash \bar{x} : \bar{A}}{\Gamma \vdash x, \bar{x} : A, \bar{A}} \qquad \frac{}{\Gamma \vdash \cdot : \cdot}$$

Here and in several other rules, *sort* is a meta-level function assigning a sort to every expression. The sort of a type is `type`, of `type` is `kind`, and of a formula is `formula`.

The `Terminates` judgment. Specificational arguments are required to be terminating, using a `Terminates` judgment. This is also used in quantifier proof rules below. Terminating terms here are just *inactive* terms I:

$$\begin{array}{rcl}
t & ::= & x \mid\mid c \mid\mid \texttt{fun}\, x(\bar{o}\,\bar{x}\ :\ \bar{A}) : T.\, t \mid\mid (t\ s\ X) \mid\mid \\
 & & \texttt{cast}\ t\ \texttt{by}\ P \mid\mid \texttt{abort}\ T \mid\mid \\
 & & \texttt{let}\ x \texttt{ = } t\ \texttt{by}\ y\ \texttt{in}\ t' \mid\mid \\
 & & \texttt{match}\ t\ \texttt{by}\ x\ y\ \texttt{with} \\
 & & \quad c_1\ \bar{s}_1\ \bar{x}_1 \texttt{ => } t_1 | \cdots | c_n\ \bar{s}_n\ \bar{x}_n \texttt{ => } t_n\ \texttt{end} \mid\mid \\
 & & \texttt{existse\_term}\ P\ t \\
 & & \\
X & ::= & t \mid\mid T \mid\mid P \\
 & & \\
A & ::= & T \mid\mid \texttt{type} \mid\mid F \\
 & & \\
T & ::= & x \mid\mid d \mid\mid \texttt{!} \mid\mid \texttt{Fun}(o\, x\ :\ A).\, T \mid\mid \langle T\ Y \rangle \\
 & & \\
Y & ::= & t \mid\mid T
\end{array}$$

Figure 1. Terms (t) and Types (T)

$$\begin{array}{rcl}
I & ::= & x \mid\mid c \mid\mid T \mid\mid P \mid\mid (c\, I_1\ \cdots\ I_n) \mid\mid \texttt{cast}\ I\ \texttt{by}\ P \mid\mid \\
 & & \texttt{fun}\, x(\bar{x}_1\ :\ A_1)\ \cdots (\bar{x}_n\ :\ A_n) : T.\, t
\end{array}$$

The class of terms which `Terminates` recognizes as terminating is expanded in Section 7 below.

Conditions on `match`. The `match` typing rule has one premise for each case of the match expression (indicated using meta-level bounded universal quantification in the premise). In each case, the two assumption variables (declared just following the "`by`" in `match`-terms) take on different classifiers. The first serves as an assumption that the scrutinee equals the pattern, and the second that the scrutinee's type equals the pattern's type. The premise requiring T to be a type is to ensure it does not contain free pattern variables. The rule also has several conditions not expressed in the figure. First, the term constructors $c_1, \cdots, c_n$ are all and only those of the type constructor d, and n must be at least one (`match`es with no cases are problematic for type computation without an additional annotation). Second, the context Δ_i is the one assigning to pairwise distinct variables $\bar{x}_i$ the types required by the declaration of the constructor c_i. Third, the type T_i is the return type for constructor c_i, where the pattern variables have been substituted for the input variables of c_i. Fourth, the type constructor is allowed to be 0-ary, in which case $\langle d\ \bar{X} \rangle$ should be interpreted here as just d. The uninformative formalization of these conditions is omitted.

4.1 Definitional Equality

Proofs, type annotations, and specificational data are of interest only for type checking, and are dropped during evaluation. OpTT's definitional equality takes this into account. It also takes into account safe renaming of variables, and replacement of defined constants by the terms they are defined to equal. Flattening of left-nested applications, and right-nested `fun`-terms and `Fun`-types is also included. More formally, definitional equality is the least congruence relation which makes (terms or types) $Y \approx Y'$ when any of these conditions holds:

1. $Y =_\alpha Y'$ (Y and Y' are identical modulo safe renaming of bound variables).
2. $Y \equiv \hat{Y}[c]$ and $Y' \equiv \hat{Y}[t_c]$, where c is defined non-recursively at the top level to equal t_c (see Section 4.4 below).

$$\frac{\Gamma(v) = A}{\Gamma \vdash v : A} \qquad \frac{\Gamma \vdash T : \texttt{type}}{\Gamma \vdash \texttt{abort}\ T : T}$$

$$\frac{\begin{array}{l} x, \bar{x} \notin \mathit{FV}(T) \\ \Gamma \vdash \bar{x} : \bar{A} \quad \Gamma, \bar{x} : \bar{A}, x\ :\ \texttt{Fun}(\bar{x}\ :\ \bar{A}).\, T \vdash t : T \end{array}}{\Gamma \vdash \texttt{fun}\, x(\bar{o}\,\bar{x}\ :\ \bar{A}) : T.\, t : \texttt{Fun}(\bar{o}\,\bar{x}\ :\ \bar{A}).\, T}$$

$$\frac{\begin{array}{l} \text{if } s = \texttt{spec}\text{, then } \texttt{Terminates}\ X \\ \Gamma \vdash t : \texttt{Fun}(x : A).\, T \quad \Gamma \vdash X : A \end{array}}{\Gamma \vdash (t\ s\ X) : [X/x]T}$$

$$\frac{\Gamma \vdash t : T_1 \quad \Gamma \vdash P : \{T_1 = T_2\}}{\Gamma \vdash \texttt{cast}\ t\ \texttt{by}\ P : T_2}$$

$$\frac{\Gamma \vdash t : A \quad \Gamma, x : A, y : \{x \texttt{=} t\} \vdash t' : T \quad x, y \notin \mathit{FV}(T)}{\Gamma \vdash \texttt{let}\ x \texttt{ = } t\ \texttt{by}\ y\ \texttt{in}\ t' : T}$$

$$\frac{\begin{array}{l} \Gamma \vdash t : \langle d\ \bar{X} \rangle \quad \Gamma \vdash T : \texttt{type} \\ \forall\, i \leq n. \\ (\Gamma, \Delta_i, x : \{t \texttt{=} (c_i\ \bar{x}_i)\}, y : \{\langle d\ \bar{X} \rangle \texttt{=} T_i\} \vdash t_i : T) \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{match}\ t\ \texttt{by}\ x\ y\ \ \texttt{with} \\ \qquad c_1\ \bar{s}_1\ \bar{x}_1 \texttt{=>} t_1 | \ldots | c_n\ \bar{s}_n\ \bar{x}_n \texttt{=>} t_n\ \texttt{end}\ :\ T \end{array}}$$

$$\frac{\begin{array}{l} \Gamma \vdash P : \texttt{Exists}(x : A).\hat{F}[x] \\ \Gamma \vdash t : \texttt{Fun}(\texttt{spec}\ x : A)(u : \hat{F}[x]).C \quad x, u \notin \mathit{FV}(C) \end{array}}{\Gamma \vdash \texttt{existse\_term}\ P\ t : C}$$

Figure 2. Term Classification

3. Nested applications and abstractions in Y and Y' flatten to the same result, as mentioned above.
4. $Y \Rightarrow Y'$, using the first-order term rewriting system of Figure 3 (where we temporarily view abstractions as first-order terms).

The rules of Figure 3 drop annotations in favor of the special constant `!`, mentioned above. There, we temporarily write P^- for a proof P which is not `!`, and similarly for T^- and A^-. The rules also operate on members of the list of input declarations in a `fun`-term, as first class expressions. Such lists can be emptied by dropping specificational inputs (hence the first rule in the figure). We temporarily consider patterns in `match` terms as applications, and hence apply the rules for rewriting applications to them. The rules are locally confluent and terminating, so we can define a function $|\cdot|$ to return the unique normal form of any expression under the rules. Note that because dropping annotations does not depend on the classification judgment, it is defined on both typeful and type-free (possibly even ill-typed) expressions.

Definitional equality is easily decided by, for example, considering the *unannotated expansions* of the expressions in question. These expansions result from replacing all constants with their definitions, then dropping all annotations, and then putting terms into an α-canonical form.

The distinction between terms, types, proofs, and formulas provides a simple principled basis for adopting different definitional equalities in different settings. Definitional equality as just defined we term *computational* definitional equality, and use it when classifying terms not inside types, proofs, or formulas. We also define a *specificational* definitional equality, used in all other situations. The difference at this point in our development is just that specificational equality drops specificationality annotations from `Fun`-

$$\begin{array}{lcl}
\texttt{fun}\,x()\ :\ T\,.\,t & \Rightarrow & t \\
\texttt{fun}\,x(\bar{x}\ :\ \bar{A})\ :\ T^-\,.\,t & \Rightarrow & \texttt{fun}\,x(\bar{x}\ :\ \bar{A})\ :\ \texttt{!}\,.\,t \\
P^- & \Rightarrow & \texttt{!} \\
(t\,T^-) & \Rightarrow & (t\ \texttt{!}) \\
(t\ \texttt{spec}\ X) & \Rightarrow & (t\ \texttt{!}) \\
(t\ \texttt{!}) & \Rightarrow & t \\
\texttt{cast}\ t\ \texttt{by}\ P & \Rightarrow & t \\
\texttt{abort}\ T^- & \Rightarrow & \texttt{abort !} \\
\texttt{existse\_term}\ P\ t & \Rightarrow & t \\
(\bar{x}\ :\ \bar{A}-) & \Rightarrow & (\bar{x}\ :\ \texttt{!}) \\
(\texttt{spec}\ \bar{x}\ :\ \bar{A}-) & \Rightarrow & \cdot
\end{array}$$

Figure 3. Dropping Annotations

types:

$$\texttt{Fun}(\texttt{spec}\ x : A).\, T \quad \Rightarrow \quad \texttt{Fun}(x : A).\, T$$

These annotations are relevant only for type checking terms and specificationality analysis, and not for reasoning. Dropping them during formal reasoning avoids some clutter in proofs, since theorems need not mention specificationality. We will put the distinction between computational and specificational definitional equality to work more crucially when we consider functional modeling in Section 8 below.

4.2 Operational Semantics

Evaluation in OPTT is call-by-value. We omit the straightforward definition of the small-step evaluation relation $\rightsquigarrow$ for space reasons. Note only that it is defined just on terms with annotations dropped. Also, we take $\rightsquigarrow$ to be small-step partial evaluation, where free (term-level) variables are considered to be values. Such variables can be introduced by the universal introduction proof rule or induction proof rule. Matching on such a variable during partial evaluation results in a stuck term.

4.3 Type Refinement

In principle, the assumption variables provided by `match`-terms are sufficient for building proofs needed to refine the types of terms (by casting) in cases. In practice, while automation cannot perform all necessary refinements (due to undecidability of type inhabitation in the presence of indexed datatypes), some automation can alleviate the burden of casts significantly. GURU implements a simple form of type refinement using first-order matching (modulo definitional equality) of the type of the pattern and the type of the scrutinee. Pattern variables occurring in the type of the pattern may be filled in by such matching, for the benefit of type checking the case associated with the pattern. Impossible cases are detected by match failure, and are neither type checked nor compiled. For example, if we analyze a scrutinee of type `<vec A (S n)>`, for vectors of `A`s of length $n + 1$, then GURU's type refinement determines that the `match`-term does not need a case for the empty vector. This is because the empty vector's type is `<vec A Z>`, and we get a match failure trying to match `Z` against `(S n)`. Other systems implement similar (or indeed, more advanced) versions of this feature.

4.4 Commands and Datatypes

Input to GURU is a sequence of commands, the most central of which are described in Figure 4. Here, G ranges over terms, types and type families, formulas, and proofs (defined in Section 4 above and 5 below). K is for kinds. Type and term constructors are introduced by the `Inductive` command, and defined constants by the `Define` command. Datatypes may be both term- and type-indexed. We additionally restrict input types to a (term) constructor for d so that they may contain d nested in type applications, but not in `Fun`-types or formulas. For meta-theoretic reasons, the input types of term constructors (that is, the types stated for inputs to the constructor) may not contain `Fun`-abstractions or `Forall`-quantifications over `type`. This may not be essential, but the current normalization argument depends on it (and it does not seem needed in practice so far). The syntax also prohibits type constructors from accepting proofs as arguments. So far we have not found a need for that feature in practice, though it could probably be added without problems. The only issue is to make sure that proofs are irrelevant if used as type indices, but the rules of Figure 3 would ensure this.

$$\texttt{Define}\ c\ :\ A\ :=\ G.$$

$$\texttt{Inductive}\ d\ :\ K\ :=\ c_1 : D_1\ |\ \ldots\ |\ c_k : D_k\,.$$

where

$$\begin{array}{lcl}
D & ::= & \texttt{Fun}(\bar{y}\ :\ \bar{A}).\langle d\,Y_1\ \ldots\ Y_n\rangle \\
K & ::= & \texttt{type}\ ||\ \texttt{Fun}(x\ :\ B).\,K \\
B & ::= & \texttt{type}\ ||\ T
\end{array}$$

Figure 4. Commands

5. Proofs and Formulas

The syntax of OPTT formulas is given in Figure 5. For compact notation, we view implications as degenerate forms of universal quantifications, and similarly conjunctions as existential quantifications. We find we do not need disjunction (not to be confused with the boolean `or` operation) for any of a broad range of program verification examples, so we have currently excluded it, for meta-theoretic simplicity. The syntax-directed classification rules for formulas are given in Figure 6. The full syntax for proofs is omitted, although a few examples are given next. There are standard logical inferences for the quantifiers, and equational inferences for proving equalities. The latter include principles of injectivity and range disjointness for term and type constructors, as well as congruence rules. There is also a construct for induction over the structure of an element of a possibly indexed datatype. The following are two important sample proof classification rules:

$$\frac{\Gamma \vdash P : \texttt{Forall}(x : A).\, F \quad \Gamma \vdash X : A \quad \texttt{Terminates}\ X}{\Gamma \vdash [P\ X] : [X/x]F}$$

$$\frac{|t|\ \rightsquigarrow\ |t'|}{\Gamma \vdash \texttt{evalstep}\ t : \{t = t'\}}$$

Untyped equations. Equations and disequations are formed between type-free terms, as well as types. Instead of allowing any untyped terms, one could require some form of approximate typing, but this is not essential nor required in practice.

Evaluation. The rule `evalstep` axiomatizes the small-step operational semantics. In practice, as in other theorem provers, higher-level tactics are needed. GURU implements several of these, discussed below.

`Terminates` and Quantifiers. `Forall`-elimination and `Exists`-introduction require the instantiating and witnessing terms, respectively, to be typed terminating expressions. Quantifiers in OPTT range over values (excluding non-terminating terms), and hence this restriction is required for soundness. In particular, `induction`-proofs establish universally quantified formulas by case analysis on the form of data in a particular datatype. This would be unsound if

$$
\begin{array}{rcl}
F & ::= & \mathit{Quant}(x\ :\ A).\,F \mid\mid \{Y_1 \overset{?}{=} Y_2\} \\[2ex]
\mathit{Quant} & \in & \{\texttt{Forall}, \texttt{Exists}\} \\
\overset{?}{=} & \in & \{\texttt{=}, \texttt{!=}\}
\end{array}
$$

Figure 5. Formulas (F)

$$\frac{\Gamma \vdash A : \mathit{sort}(A) \quad \Gamma, x\ :\ A \vdash F : \texttt{formula}}{\Gamma \vdash \mathit{Quant}(x\ :\ A).\,F : \texttt{formula}}$$

$$\frac{}{\Gamma \vdash \{t_1 \overset{?}{=} t_2\} : \texttt{formula}}$$

$$\frac{}{\Gamma \vdash \{T_1 \overset{?}{=} T_2\} : \texttt{formula}}$$

Figure 6. Formula Classification

such a formula could be instantiated by a term which could fail to normalize to a piece of data in that datatype.

Induction. `Induction`-proofs are are similar to a combination of terminating recursive `fun`-terms and `match`-terms. The syntax is:

$$\texttt{induction}(\bar{x}\ :\ \bar{A})\texttt{by}\ x\ y\ z\ \texttt{return}\ F$$
$$\texttt{with}\ c_1\ \bar{x}_1\ \texttt{=>}\ P_1 | \cdots | c_n\ \bar{x}_n\ \texttt{=>}\ P_n\ \texttt{end}$$

The third assumption variable (z) is bound in the cases for the induction hypothesis. The first two play similar roles as in `match`-terms. The last classifier in the list $\bar{A}$ is required to be a datatype (i.e., of the form $\langle d\ \bar{Y}\rangle$). The last variable in the list is thus the parameter of induction. Earlier parameters may be needed due to dependent typing. The classifier for the proof is then of the form $\texttt{Forall}(\bar{x}\ :\ \bar{A}).F$.

5.1 Evaluation Tactics

While the core proof language just described is sufficient in theory, in practice one needs tactics, as in other theorem provers. Theorem provers often provide incomplete or non-terminating tactics, though of course not unsound ones. GURU implements several tactics for equational reasoning. The simplest is `eval`, for evaluating a term to a normal form:

$$\frac{|t|\ \leadsto^{!}\ |t'|}{\Gamma \vdash \texttt{eval}\ t : \{t \texttt{=} t'\}}$$

Similarly, we have `evalto`, which may stop before a normal form is reached:

$$\frac{|t|\ \leadsto^{*}\ |t'|}{\Gamma \vdash \texttt{evalto}\ t\ t' : \{t \texttt{=} t'\}}$$

Most frequently used in practice is `join`, for joining terms at a normal form (recall that $\leadsto^{!}$ is standard notation from term rewriting theory for reduction to a normal form):

$$\frac{|t|\ \leadsto^{!}\ t'' \qquad |t'|\ \leadsto^{!}\ t''}{\Gamma \vdash \texttt{join}\ t\ t' : \{t \texttt{=} t'\}}$$

The most sophisticated tactic is `hypjoin` (Petcher 2008). The syntax for this tactic is the following:

$$\texttt{hypjoin}\ t\ t'\ \texttt{by}\ P_1\ \cdots P_n\ \texttt{end}$$

This is like `join`, in that it tries to join terms t and t'. It is given proofs $P_1, \ldots, P_n$, however, each of which should prove a (possibly non-normalized) ground equation. The tactic then tries to join the terms modulo those equations. This essentially requires computing a congruence closure modulo evaluation. In (Petcher 2008), `hypjoin` is proved sound and also, under some conditions on termination of the terms involved, complete. In practice, `hypjoin` is very useful for proving equations without giving the kind of detailed proofs which are otherwise required. The only current drawback is that it is often significantly slower to check a `hypjoin` proof than to check the sort of detailed proof it is capable of finding. One might conjecture that this is due to the need (for completeness) to perform exhaustive interreduction of the equations, when many of those reductions are not needed in the proof that is found. This remains to be further investigated, however.

All the above evaluation tactics currently rely on undecidable side conditions about evaluation, and may fail to terminate. If they terminate, however, a proof using just `evalstep` and basic equational reasoning principles can in principle be reconstructed, though this is not currently implemented in GURU.

6. Simple Examples

We consider here two simple examples, which have been machine checked using the GURU implementation of OPTT. The first could be done in type theories like COQ's, albeit with more work (in a precise sense described below). The second cannot be done directly, due to the presence of a function which truly can diverge on some inputs. It could be handled by indirect means, for example by encoding it as a relation. But such means incur their own costs: one can no longer use COQ's built-in partial evaluation with a function encoded as a relation.

6.1 Associativity of Append on Lists with Length

Internally verifying that the length of appended lists is equal to the sum of their lengths is standard for dependently typed programming. Externally verifying associativity of such a function is not. Such reasoning is possible in systems like COQ, but generally requires the use of an additional axiom expressing some form of proof irrelevance (Hofmann and Streicher 1998; The Coq Development Team 2004). This case does not require such an axiom, but, as typically implemented, does require tedious manipulations of casts. Indeed, it is often remarked that external reasoning about dependently typed functions is tedious in COQ.

For our GURU implementation, we first declare a datatype of lists with length, assuming a standard definition of the datatype `nat` for unary natural numbers. The type `<vec A n>` is inhabited by all and only (finite) lists of elements of type `A` of length `n`. The `vecn` constructor creates a list of length zero (“`Z`”), and `vecc` one of length `(S n)` from a list of length `n`.

```
Inductive vec : Fun(A:type)(n:nat). type :=
  vecn : Fun(A:type). <vec A Z>
| vecc : Fun(A:type)(n:nat)(a:A)
           (l:<vec A n>).<vec A (S n)>.
```

We may now define a recursive function `append` with the following type. This type states that the length of the output list is the sum (assuming a standard definition of `plus`) of the lengths of the input lists. It also records that the lengths are specificational data. Specificational data analysis enforces statically that computational results cannot depend on these lengths, but only the lists themselves.

```
Fun(A:type)(spec n m:nat)
   (l1 : <vec A n>)(l2 : <vec A m>).
   <vec A (plus n m)>
```

The code for `append` is the following, where `P1` and `P2` are short equational proofs omitted here:

```
fun append(A:type)(spec n m:nat)
          (l1 : <vec A n>)(l2 : <vec A m>) :
          <vec A (plus n m)>.
  match l1 by u v with
    vecn A' => cast l2 by P1
  | vecc A' n' x l1' =>
    cast
      (vecc A' (plus n' m) x
        (append A' n' m l1' l2))
    by P2
  end
```

The expected return type of the function is `<vec A (plus n m)>`, but in each case without the casts we have something with a provably equal but not definitionally equal type. So in each case, a cast is used to change the type. The proof `P1` proves that { `m = (plus n m)`}. The proof `P2` proves that { `(S (plus n' m)) = (plus n m)` }. These proofs uses the assumption `v` that the type of `l1`, namely `<vec A n>`, is equal to the type of the pattern, namely `<vec A Z>` in the first case, and `<vec A (S n')>` in the second. From these, using injectivity of `vec`, we can derive {`n = Z`} and {`n = (S n')`}, respectively; from which the equation between `n` and `(plus n m)` follows in each case.

The statement of associativity is the following:

```
Forall(A:type)(n1 n2 n3 : nat)
      (l1 : <vec A n1>)
      (l2 : <vec A n2>)
      (l3 : <vec A n3>).
  { (append (append l1 l2) l3) =
    (append l1 (append l2 l3)) }
```

Since for `append`, the lengths `n1`, `n2`, and `n3` are specificational data, they are dropped in type-free positions. Hence, the equation to be proved does not mention those lengths. In type theories like COQ's or Epigram's, in contrast, the equation to be proved must be typed, and so must mention the lengths:

```
{ (append (plus n1 n2) n3
     (append n1 n2 l1 l2) l3) =
  (append n1 (plus n2 n3)
       l1 (append ! n2 n3 l2 l3)) }
```

In fact, since the two sides of this latter equation have, respectively, types `<vec A (plus (plus n1 n2) n3)>` and `<vec A (plus n1 (plus n2 n3))>`, even stating this theorem requires heterogeneous equality. The proof of the equality must contain in it a proof of associativity of addition. In contrast, in GURU, since the lengths are dropped, the proof of associativity is just as for `append` on lists without length. The proof does not require associativity of `plus`.

6.2 Untyped Lambda Calculus Interpreter

We internally verify that a simple call-by-value interpreter for the untyped lambda calculus maps closed terms to closed terms. The datatype for lambda terms t is indexed by the list of t's free variables. Using explicit names for free variables is adequate for our purposes here. We take names to be natural numbers. The datatype of terms is the following:

```
Inductive lterm : Fun(l:<list nat>).type :=
  var : Fun(v:nat).
          <lterm (cons nat v (nil nat))>
| abs : Fun(a:nat)(l:<list nat>)(b:<lterm l>).
          <lterm (removeAll nat eqnat a l)>
| app : Fun(l1 l2:<list nat>)
          (x1:<lterm l1>)(x2:<lterm l2>).
          <lterm (append nat l1 l2)>.
```

Here, `removeAll` removes all occurrences of an element from a list, and `append` appends lists (without length). Note that this example, implemented before implementation of the specificationality analysis described above was complete, does not use specificational data. It should be possible to make the lists of nats that are given as arguments to the constructors for `lterm` specificational.

The crucial helper function is for substitution of a closed term `e2` for a variable `n` into an open term `e1`, with list of free variables `l`. This substitution function has the following type:

```
Fun(e2:<lterm (nil nat)>)(n:nat)
   (l:<list nat>)(e1:<lterm l>).
   <lterm (removeAll nat eqnat n l)>
```

Note that here we are internally verifying a certain relationship between the sets of free variables of the input terms and the output term. Internally, this code uses several (external) lemmas about `removeAll`. Where substitution enters another lambda abstraction, a commutativity property is required, saying that removing x and then removing y results in the same list as removing y and then x. Using substitution, we can implement β-reduction for closed redexes in the interpreter. The interpreter then has the following type, which verifies internally that evaluation of a closed term, if it terminates, produces a closed term:

```
Fun(e1:<lterm (nil nat)>).<lterm (nil nat)>
```

We also externally verify that if this interpreter terminates, then its result is a lambda abstraction. Here, we do not bother to track the fact that the list of free variables in the resulting abstraction is empty (this could be easily done).

```
Forall(e1 e2:<lterm (nil nat)>)
      (p:{(lterm_eval e1) = e2}).
  Exists(a:nat)(l:<list nat>)(b:<lterm l>).
    { e2 = (abs a b) }
```

The proof of this property relies on a principle of computational induction, for reasoning by induction on the structure of a computation which is (assumed here to be) terminating, namely `(lterm_eval e1)`.

7. Termination Casts

Universal elimination and existential introduction require terminating terms, which up until now have been taken to be just terms already in normal form. This turns out to be too restrictive in practice, as we now explain. Suppose we want to instantiate a universal using a non-constructor term $(f\ \bar{a})$, where for simplicity suppose $\bar{a}$ are constructor terms. Using the proof rules given above, one would first prove totality of f: for all inputs $\bar{x}$, there exists an output r such that $(f\ \bar{x}) = r$. Instantiating $\bar{x}$ with $\bar{a}$ and then performing existential elimination will provide a variable r together with a proof u that $(f\ \bar{a}) = r$. Now the original universal instantiation can be done with r, translating between r and $(f\ \bar{a})$ as necessary using equational reasoning and u. If this strikes the reader as somewhat tedious, that is indeed the authors' experience. Matters are even worse with nested non-constructor applications, where the process must be repeated in a nested fashion.

To improve upon this, we extend `Terminates` from constructor terms to provably total terms, as follows. We introduce a new term construct of the form `terminates` t `by` P. This is a termination cast. Where a type cast changes the type checker's view of the type of a term, a termination cast changes its view of the termination behavior of a term. `Terminates` is extended to check that P either proves t is equal to a constructor term, or else proves totality of the head (call it f) of t, in the sense mentioned above. `Terminates` still must check that subterms of applications are terminating, even if the head is. Note that Terminates is now contextual, since the proof P may use hypotheses from the context. The basic design of OpTT makes this addition straightforward, since termination casts are computationally irrelevant. We extend our definitional equalities by dropping terminates-annotations:

$$\texttt{terminates}\ \mathrm{t}\ \mathrm{by}\ \mathrm{P} \quad \Rightarrow \quad t$$

Termination casts may be used in universal instantiation and existential introduction, but are eliminated by definitional equality during equational reasoning.

8. Functional Modeling, Ownership Annotations

Inspired by a suggestion of Swierstra and Altenkirch, GURU supports non-functional operations like destructive updates and input/output via functional modeling (Swierstra and Altenkirch 2007). The basic idea is to define a functional model of the non-functional operations. This model can be used for formal reasoning. It is replaced during compilation by its non-functional implementation, which must be trusted correctly to implement the functional model. To ensure soundness, usage of the functional model in code is linearly restricted. Swierstra and Altenkirch propose using monads for this. Here, we use uniqueness types (Barendsen and Smetsers 1993). Types and type families can be designated as *opaque*, in which case any functions which perform case analysis on them must be marked specificational when they are defined at the top level in GURU. Functions marked specificational must be replaced during compilation.

Previously, fun-terms could specify that arguments are specificational or not. Now, we extend these annotations to include ownership annotations unique and unique_owned. Inputs marked unique must be consumed by the function exactly once, and may not have their reference counts incremented. Those marked unique_owned may be inspected but must not be consumed or have their reference counts incremented. Term constructors may take unique (but not unique_owned) arguments. Applications of such to unique expressions become unique as well, consuming the resource. Functions marked specificational need not obey uniqueness requirements, since they will be replaced by trusted non-functional implementations. A simple static analysis ensures correct resource usage.

The distinction mentioned above (Section 4.1) between computational and specification definitional equality is here crucial. In specificational functions, we use the specificational equality, which takes definitions of opaque types into account. In computational functions, we use the computational equality, which does not. This allows formal reasoning to make use of the function model, while prohibiting computational functions from violating the abstraction boundary imposed by opacity. For example, if 32-bit words are modeled as vectors of booleans of length 32, then operations on vectors must not in general be applied to words; only those marked as specificational, which will be replaced during compilation. Ownership annotations are dropped in the specificational equality, reducing clutter during external reasoning.

9. Reference Counting and Compilation

Since all data in OpTT are inductive, the data reference graph is truly acyclic. So GURU's compiler (to C code) implements memory reclamation using reference counting, instead of garbage collection. Reference counting is sometimes criticized as imposing too much overhead, due to frequent increments and decrements. GURU puts this under the control of the programmer via explicit incs and decs. But GURU also provides ownership annotations to reduce the need for these. Function inputs may be marked as owned by the calling context. To consume them, the function must do an inc. But just to inspect them by pattern matching does not require an inc, and the function may notdec an owned input. The static analysis mentioned above for tracking uniqueness also ensures correct reference counting. Pattern matching consumes unowned resources. Functions and flat inductive data like booleans are not tracked. The former is sound here because GURU does not implement closure conversion. Closures may be implemented by hand, thanks in part to OpTT's System-F-style polymorphism.

When the reference count of a piece of data falls to zero, it is, of course, time to reclaim the memory associated with that data. As shown in the empirical results of the larger case study below, much high-performance allocation can be achieved using a custom allocation scheme rather than just the standard malloc and free library functions. GURU's approach is the following. We keep a distinct free list for every term constructor. Suppose it is time to reclaim the memory for a data element of the form $(c\ \bar{V})$. Then we simply add that data element to the free list for c, without changing the reference counts for the subdata $\bar{V}$. Subsequently, when it is time to allocate a new piece of data built with constructor c, we pull $(c\ \bar{V})$ off the free list. At that point, we decrement the reference counts for the subdata $\bar{V}$. In addition to being very fast in practice (on the benchmarks tested so far), this scheme provides time-bounded memory management operations. The time required to free a data item is a small constant, and the time to allocate a new item is bounded by the number of arguments possible to a constructor (since we must decrement the reference counts for subdata $\bar{V}$ if we are returning an element $(c\ \bar{V})$ from the free list for c). The time-bounded nature of this memory management scheme makes it much more attractive for real-time systems, for example, where achieving real-time garbage collection is a subject of ongoing research.

Note that the basic design idea of OpTT again helps here, since we make increments and decrements (of terminating terms) computationally irrelevant via definitional equality. They need not be considered during formal reasoning.

One final note about reference counting and compilation concerns polymorphism. GURU implements polymorphism in the style of System F by passing type representations at run-time. The type representation here are very simple. We just associate an integer with every type constructor. We do not need a representation for every application of a type constructor. We must also associate integers with function types declared in the C source, which arise from flattening nested function types of GURU (since C allows definitions of function types, but not nested ones). These type representations are passed whenever GURU code passes a type as an argument. So they are also passed to term constructors, and hence stored in polymorphic data structures. They have just one purpose, arising from the following situation. Suppose we have a polymorphic term constructor like cons for (homogeneous) polymorphic lists. So we have terms of the form $(\texttt{cons}\ A\ a\ l)$, where A is a type, $a : A$, and l is the rest of the list. When it is time to decrement the reference count for a, this must be done in a type-dependent way. If a is an untracked piece of data (an element of a flat inductive type or a function), then we should do nothing, since such data do not have a reference count to decrement. Otherwise, we should decrement the reference count. If that falls to zero, we must put a onto the free list for whichever constructor a is built with. So we use the type representation for A as an index into a table of decrement functions, thus dispatching to the appropriate function for a. The corresponding dispatch occurs also when it is time to increment the reference count for a piece of data of type A, with A a type variable.

10. Case Study: Incremental LF

This section describes a larger case study carried out in Guru, in the domain of efficient proof checking. In automated theorem proving, the complexity of solver implementations limits trustworthiness of their results. For example, modern SMT (Satisfiability Modulo Theories) solvers typically have codebases around 50k-100kloc C++ (e.g., CVC3 (Barrett and Tinelli 2007)). One method to help catch solver errors and to export results to skeptical interactive theorem provers is to have the solvers emit proofs. Independent checking of the proofs by a much smaller and simpler checker can confirm the solvers' results. Efficient and flexible proof checking for tools like SMT solvers is a subject of current interest in the SMT community (e.g., (Moskal 2008)). A proposal of the first author is to use an extension of the Edinburgh Logical Framework (LF) as the basis for efficient and flexible proof checking for SMT (Stump and Oe 2008; Harper et al. 1993). LF is a dependent type theory with support for higher-order abstract syntax, used previously in proof-carrying code and related applications (e.g., (Appel 2001; Necula 1997)). In LF encoding methodology, proof checking in an object logic is reduced to type checking in LF. To handle large proofs from SMT solvers, several optimizations for LF type checking have been proposed, including *incremental checking*.

10.1 Incremental LF Type Checking

Incremental checking intertwines parsing and type checking for LF (Stump 2008). The goal is to avoid creating abstract syntax trees (ASTs) in memory whenever possible. ASTs must be created for expressions which will ultimately appear in the type of a term, but others need not. This gives rise to one pair of modes, namely creating vs. non-creating. Standard bi-directional type checking for canonical forms LF gives rise to an orthogonal pair of modes, namely type synthesizing (computing a type for a term in a typing context) vs. type checking (checking that a term has a given type in a typing context) (Watkins et al. 2002; Pierce and Turner 1998). To check a term, we are initially in non-creating mode. When we encounter an application with head term of type $\Pi x : A. B$, where x is free in B, we must switch to creating mode to check the argument term. If x is not free in B, we may remain in non-creating mode, thus avoiding building an AST for the argument. An implementation of incremental checking in around 2600 lines of C++ has been evaluated on benchmark proofs generated from a simple quantified boolean formula (QBF) solver (Stump 2008). The results show running times faster than those previously achieved by *signature compilation*, where a signature is compiled to an LF checker customized for checking proofs in that signature (Zeller et al. 2007). Implementing the incremental type checker in C++ is quite error-prone, due to lack of memory safety in C++, and the dependence of outputs on requested checking modes.

10.2 Incremental Checking in Guru

An incremental LF checker called Golfsock has been implemented in Guru, where we internally verify two properties. First, mode usage is consistent, in the sense that if the core checking routine is called with a certain combination of modes (from the orthogonal pairs checking/synthesizing and creating/non-creating), then the appropriate output will be produced: the term, iff in creating mode; and its type, iff in synthesizing mode. Second, whenever a term is created, there is a corresponding typing derivation for it in a declarative presentation of LF. Golfsock is somewhat unusual compared with related examples (e.g., (Urban et al. 2008)), due to the need to use more efficient data structures than typically used in mechanized metatheory. We consider a few of these data structures next.

10.3 Machine Words for Variable Names

Golfsock uses 32-bit machine words for variable names. An earlier version used unary natural numbers for variables, but performance was poor, with profiling revealing 97% of running time on a representative benchmark going to testing these for equality. Replacing unary natural numbers with 32-bit words resulted in a 60x speedup on that benchmark. But using 32-bit words for variable names requires significant additional reasoning in Golfsock. The reason is that capture-avoiding substitution relies on having a strict upper bound for the variables (bound or free) involved in the substitution. This strict upper bound is used to put the term into α-canonical form during substitution: all bound variables encountered are renamed to values at or above the upper bound, thus ensuring that free variables are not captured. We must maintain the invariant that new upper bounds produced by functions are always greater than or equal to the initial upper bounds. This requires incrementing of 32-bit words and inequality, as well as associated lemmas. Fortunately, the Guru standard library includes an implementation of bitvectors (as vectors of booleans), with an increment function, functions mapping to and from unary natural numbers, and appropriate lemmas about these. These are specialized to vectors of length 32 for machine words. For Golfsock, the most critical of these are `word_inc`, which increment a 32-bit word, reporting if overflow occurred; and the following lemma stating that if incrementing word `w` produces `w2` without overflow (`ff` is boolean false), then mapping `w2` to a unary natural number gives the successor of the result of mapping `w`.

```
Define word_to_nat_inc2
   : Forall(w w2:word)
           (u : { (word_inc w) =
                  (mk_word_inc_t w2 ff)}).
      { (S (word_to_nat w)) = (word_to_nat w2) }
```

Note that `mk_word_inc_t` is a term constructor used to pass back both the incremented word (its first argument) and a boolean telling whether or not overflow occurred (its second argument). To use this lemma, Golfsock aborts if overflow occurs. Provisions are included to reset the upper bound in non-creating, checking mode, where this is proven sound; and overflow does not occur in any benchmarks tested. A more robust solution, of course, is to use arbitrary precision binary numbers. Implementation of these is in progress but currently not available.

Following the methodology described in Section 8, the `word` datatype is treated as opaque, with the critical computational operations on words replaced during compilation. The fact that these replacements are functionally equivalent to the operations as modeled in Guru is unproven and must be trusted. Fortunately, there are just three such operations used in Golfsock: creating the word representing 0, incrementing a word with overflow testing, and testing words for equality. These total just 8 lines of C code.

10.4 Tries and Character-Indexed Arrays

A trie is used for efficiently mapping strings for globally or locally declared identifiers to variables (32-bit words) and the corresponding LF types. Tries are implemented in the standard library with the following declaration:

```
Inductive trie : Fun(A:type).type :=
  trie_none : Fun(A:type).<trie A>
| trie_exact : Fun(A:type)(s:string)(a:A).<trie A>
| trie_next : Fun(A:type)(o:<option A>)
                 (unique l:<charvec <trie A>>).
              <trie A>.
```

The first constructor is for an empty trie, the second for a trie mapping just one string to a value, and the third for a trie mapping multiple strings to values. The second and third overlap in usage: we can map a single string to a value using one `trie_exact` or a nesting of `trie_next`s. This `trie_next` uses an opaque datatype `charvec` for character-indexed arrays, where characters are 7-bit words (for ASCII text only). These arrays are modeled functionally as vectors of length 128. We statically ensure that array accesses are within bounds, since the vector read function requires a proof of this. Destructive array update is supported with uniqueness types, ensuring access patterns consistent with destructive modification. During compilation, the functional model is replaced by an implementation with actual C arrays, and constant-time read and write operations. Operations implemented on tries include insertion, lookup, and removal, as well as a function `trie_interp` which maps a trie to a list of (key,value) pairs.

The fact that `trie_next` contains a character-indexed array of tries poses a challenge for proving theorems about tries. The problem is that trie operations access subtries of a trie `T` via an array read. In the functional model, the resulting subtrie is not a structural subterm of `T`, and so proof by induction on trie structure cannot apply an induction hypothesis to the subtrie. This problem may not seem difficult: informal reasoning can easily get around this problem using instead complete induction on the size of the trie. But how can we write a provably total function to compute the size of a trie? Such can certainly be implemented in Guru, but to prove it total we are back to the same problem it was introduced to solve: the natural totality proof proceeds essentially by induction on the structure of the trie. Indexing tries by their size does not help, since the character-indexed array type is homogeneous: it cannnot store subtries of different sizes, if the size is part of the type of tries. One could get around this problem by indexing tries by an upper bound on their sizes, but then inserting a bigger subtrie into the array would require weakening the upper bounds for all the other tries. In a language with good support for eliminating coercion functions (like weakening the upper bound on a trie's size) during compilation, this would solve the problem. Such a feature is perhaps non-trivial to implement, and Guru does not have it presently.

A different, easy solution is enabled by the separation of terms and proofs in OpTT. We introduce a specificational construct `size t` to compute the size of any value. Functions are assigned size 0, while constructor terms are assigned the successor of the sizes of their subterms. The evaluation rules of the theory are extended appropriately. We may now prove properties about trie operations by complete induction on trie size computed by this construct. OpTT's design allows us to make such an addition without needing to reconsider any meta-theory: neither the logical consistency argument (since `size` may not be applied to proofs) nor type soundness for term reduction (since `size` is purely specificational).

As a performance benchmark, a program to histogram the words in ASCII text was implemented in both Guru and OCaml version 3.10.1. Runtimes are indistinguishable with array-bounds checking on or off in the OCaml version. Note that array accesses are statically guaranteed to be within bounds in the Guru version. The same data structures, particularly mutable tries, and algorithms were implemented in each. Counting the number of times the word "cow" occurs in an English translation of "War and Peace" (it occurs 3 times) takes 3.7 seconds with the OCaml version on a standard test machine, and 1.5 seconds with the Guru version. Disabling garbage collection in OCaml drops the runtime to 1.2 seconds. While hardly conclusive, this experiment supports the hypothesis that programmer-controlled reference counting may not be inferior to garbage collection, at least for some applications. This is consistent with the results of a thorough study showing that garbage collection may be significantly slower than more fine-grained memory management schemes in memory-constrained settings (Hertz and Berger 2005).

benchmark	**size (MB)**	C++ impl	Golfsock	Twelf
cnt01e	2.6	0.9	1.3	9.9
tree-exa2-10	3.1	1.1	1.6	12.6
cnt01re	4.6	1.7	2.3	149.5
toilet_02_01.2	11	4.0	5.8	809.8
1qbf-160cl.0	20	6.9	8.6	timeout
tree-exa2-15	37	13.7	20.7	timeout
toilet_02_01.3	110	40.4	65.8	timeout

Figure 7. Checking Times in Seconds for QBF Benchmarks

10.5 Statistics

The code for the central `check` routine is around 1100 lines. Its size would make it challenging to reason about externally, so verifying it internally with dependent types seems the right choice. Golfsock proper is around 4000 lines of code and proofs, resting upon files from the Guru standard library totally an additional 6700 lines, mostly of proofs. The Guru compiler produces 9000 lines of C for Golfsock. A number of lemmas remain to be proved. Even so, they are more trustworthy then the several thousand lines of complex C++ code of the first author's original unverified incremental checker. This increase in trustworthiness can be confirmed anecdotally. The first author encountered just a couple of relatively benign bugs while developing it (related to properties not selected to be verified), in contrast to a long and laborious debugging effort needed for the original unverified implementation.

10.6 Empirical Results

Figure 7 gives empirical results comparing the original C++ implementation ("C++ impl") with Golfsock, and also Twelf (Pfenning and Schürmann 1999). The primary usage of Twelf is for machine-checked meta-theory (e.g., (Lee et al. 2007)), not checking large proof objects. Twelf is included here as a well-known LF checker not written or co-written by the first author. The benchmarks used are the QBF ones mentioned above, originally considered in the work on signature compilation (Zeller et al. 2007). Note that while the C++ checker has support for a form of term reconstruction (also known as implicit arguments), Golfsock does not, and hence we use the fully explicit form of these benchmarks. A timeout of 1800 seconds was imposed. The results show Golfsock is around 30% slower than the C++ version. We may consider this a good initial result, particularly since the C++ version implements many optimizations not supported in Golfsock. For example, the C++ version implements a form of delayed substitution, while Golfsock substitutes eagerly. Each such optimization which the C++ implementation can include at no (initial) cost would need to be verified in the Golfsock version, with respect to declarative LF typing. Memory usage, not reported in the table, is comparable in the C++ version and Golfsock. The version of Golfsock used for Figure 7 uses the custom memory management scheme described in Section 9 above. Figure 7 compares this version with a version which instead uses `malloc` and `free` for allocation and deallocation of memory. We see that on average that Golfsock with custom allocation is a bit more than 4 times faster than Golfsock using `malloc` and `free`. More results are not reported due to the `malloc`/`free` version exhausting memory on larger examples. Debugging with the standard memory debugging tool Valgrind does not reveal any memory leaks, so this behavior requires further investigation.

benchmark	GOLFSOCK `malloc`	GOLFSOCK `custom`
cnt01e	5.3	1.3
tree-exa2-10	6.7	1.6
cnt01re	9.7	2.3

Figure 8. Comparing `malloc`/`free` with Custom Memory Management

11. Conclusion

The GURU verified programming language provides a powerful language for implementing dependently typed functional programs and proving properties about them. The core design ideas of OPTT are put to good use in supporting specificational data, reference counting, and functional modeling with linear types. Operationally irrelevant annotations are dropped from programs during theorem proving, thus reducing the burden of proof for programmers; and similarly during compilation to efficient C code. Reference counting, particularly using annotations to reduce the number of increments and decrements, shows promise in this setting, where the reference graph is acyclic. The case study and empirical evaluation of the incremental LF checker GOLFSOCK demonstrates that GURU can be applied to build and verify efficient and realistic programs.

Acknowledgements: Thorsten Altenkirch and anonymous POPL 2009 reviewers for detailed and helpful comments on an earlier draft; Daniel Tratos and Henry Li for additions to the GURU standard library; and the NSF for support under award CCF-0448275.

References

A. Appel. Foundational Proof-Carrying Code. In *16th Annual IEEE Symposium on Logic in Computer Science*, 2001.

E. Barendsen and S. Smetsers. Conventional and Uniqueness Typing in Graph Rewrite Systems. In *Proc. 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 41–51. Springer-Verlag, 1993.

C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, pages 298–302. Springer-Verlag, 2007.

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

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.

Matthew Hertz and Emery D. Berger. Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. In *Proc. 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 313–326. ACM, 2005.

M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In G. Sambin, editor, *Twenty-five years of constructive type theory*, pages 83–111. Oxford: Clarendon Press, 1998.

D. Lee, K. Crary, and R. Harper. Towards a Mechanized Metatheory of Standard ML. In *Proc. 34th ACM Symposium on Principles of Programming Languages*, pages 173–184. ACM Press, 2007.

D. Licata and R. Harper. A Formulation of Dependent ML with Explicit Equality Proofs. Technical Report CMU-CS-05-178, Carnegie Mellon University School of Computer Science, December 2005.

C. McBride and J. McKinna. The View from the Left. *Journal of Functional Programming*, 14(1), 2004.

M. Moskal. Rocket-Fast Proof Checking for SMT Solvers. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

A. Nanevski and G. Morrisett. Dependent Type Theory of Stateful Higher-Order Functions. Technical Report TR-24-05, Harvard University, 2005.

G. Necula. Proof-Carrying Code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

U. Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

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

A. Petcher. Deciding Joinability Modulo Ground Equations in Operational Type Theory. Master's thesis, Washington University in Saint Louis, May 2008. Available from `http://cl.cse.wustl.edu`.

F. Pfenning and C. Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

Benjamin C. Pierce and David N. Turner. Local type inference. In *25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–265, 1998.

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

A. Stump. Proof Checking Technology for Satisfiability Modulo Theories. In A. Abel and C. Urban, editors, *Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.

A. Stump and D. Oe. Towards an SMT Proof Format. In C. Barrett and L. de Moura, editors, *International Workshop on Satisfiability Modulo Theories*, 2008.

A. Stump and E. Westbrook. A Core Operational Type Theory. Under review, available from `http://www.cs.uiowa.edu/~astump`.

W. Swierstra and T. Altenkirch. Beauty in the Beast. In *Haskell Workshop*, 2007.

The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version V8.0*, 2004. http://coq.inria.fr.

C. Urban, J. Cheney, and S. Berghofer. Mechanising the Metatheory of LF. In *Proc. of the 23rd IEEE Symposium on Logic in Computer Science*, pages 45–56. IEEE Computer Society, 2008.

K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A Concurrent Logical Framework I: Judgments and Properties. Technical Report CMU-CS-02-101, Carnegie Mellon University, 2002.

M. Zeller, A. Stump, and M. Deters. Signature Compilation for the Edinburgh Logical Framework. In C. Schürmann, editor, *Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*, 2007.

D. Zhu and H. Xi. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, pages 83–97, Long Beach, CA, January 2005. Springer-Verlag LNCS vol. 3350.