Verified Programming in Operational Type Theory

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

Morgan Deters
LSI, Universitat Politècnica de Catalunya
mdeters@cse.wustl.edu

Todd Schiller
Timothy Simpson
Edwin Westbrook
CSE, Washington University in St. Louis
{tws2,tas5,emw1}@cec.wustl.edu

Abstract

An Operational Type Theory (OpTT) is developed based on a theory of operational joinability of untyped terms. The theory accommodates functions which might diverge or abort on some inputs, while retaining decidability of type checking and logical consistency. For the latter, OpTT distinguishes proofs from programs, and formulas from types. Proofs and other computationally irrelevant annotations, including data marked as specificational, are dropped during formal reasoning. This greatly simplifies verification problems by removing the need to reason about types when reasoning about programs. Indeed, terms with provably disequal types can be shown equal in OpTT. OpTT is realized in a tool called GURU, featuring a type/proof checker and a compiler to C code. The theory easily accomodates extensions such as uniqueness and ownership types, used to track program resources. Several case studies are discussed, including ones based on mutable tries.

1. Introduction

Type theories based on the Curry-Howard isomorphism are of continuing appeal as providing a unified formalism for writing and verifying programs. One well-known drawback is that partial functions must be accommodated with care, for two reasons. First, naively allowing diverging terms renders the logic unsound. Second, for type theories with a definitional equality which includes computation, diverging terms can render type checking undecidable.

The present work is based on the observation that three roles for evaluation in type theory may be distinguished, which are commonly unified, particularly under the Curry-Howard isomorphism. These are evaluation for program execution, evaluation for proof normalization, and evaluation for definitional equality. Distinguishing these roles for evaluation allows much greater control over the meta-theory of the system. By devising different classification (type) systems for programs, proofs, and their classifiers, we can vary the complexity of program execution, proof normalization, and definitional equality independently. This paper presents one system based on this idea, called Operational Type Theory (OpTT). This system allows possibly diverging or finitely failing programs, while keeping the logic first order and definitional equality decidable. The focus here is on practical program verification. For formalized mathematics, a higher-order logic and richer definitional equality might be more appropriate, and could possibly be developed similarly to OPTT.

Having separated proofs and programs, the critical technical challenge is to allow them nevertheless to interact. The goal is to support a combination of internal and external verification (Altenkirch 1996). With external verification, proofs prove specificational statements about the observational behavior of programs. With internal verification, specifications are expressed through rich typing of the programs, typically using dependent function types and indexed datatypes. Each style has its advantages and disadvantages: the former is more flexible, since additional properties can be proved after coding without modifying the program; while the latter is closer to existing programming practice. The combination of internal and external verification leads to technical puzzles, such as how to state equalities between terms with provably but not definitionally equal types (Hofmann and Streicher 1998). One answer is heterogeneous equality, another is contextual definitional equality (Blanqui et al. 2008; McBride 1999).

OPTT combines internal and external verification in a different way by taking an untyped propositional equality on type-free terms, with all typing annotations dropped. Examples of such annotations are proofs in explicit casts, which are used to establish type equivalences beyond definitional equality; and also data marked as specificational. Such data have been dropped in previous work on compilation of dependently typed programs (Brady et al. 2003). OPTT takes this a step further, by allowing external reasoning in the theory about such type-free terms. The practical benefits of this are significant, since it is no longer necessary to reason about the types of terms when reasoning about the terms. Indeed, terms can be proved equal which have provably disequal types. An example of this is given in Section 5.2 below. Provable equality soundly captures joinability of untyped terms in the call-by-value operational semantics of the language, thus giving OPTT its name.

Primary Contributions. The central design idea developed in OPTT is its untyped operational equality, based on a principled separation of proofs and programs. The latter allows support for diverging terms while keeping a consistent logic and decidable definitional equality. Dropping annotations when reasoning about terms – indeed, as part of definitional equality – turns out to provide a very robust basis for extensions to the theory. Extensions discussed in this paper include termination casts (like type casts but used to show the type checker that a term is terminating); and ownership and uniqueness annotations, which allow sound functional modeling of non-functional operations like destructive updates and input/output. OPTT with these extensions has been implemented in a tool called GURU, including a type/proof checker and a compiler to C. Using this tool, several examples are pre-

sented which are challenging or impossible for other type theories (Section 5). The meta-theory of OPTT is considered, in particular cut elimination and type soundness (Sections 9 and 10). A larger case study is described (Section 12), namely an implementation called GOLFSOCK of an optimized type checker for the Edinburgh Logical Framework (Harper et al. 1993). We begin by considering additional related work, and then present the syntax, typing rules, and operational semantics of terms (Section 3). GURU and the code and proofs for the examples are freely available at `cl.cse.wustl.edu/guru/`.

2. Related Work

Intensional Type Theory. One approach to accommodating general recursive programs in intensional type theory is to require such programs to take an additional input, which restricts the program to a subset of its nominal domain on which it is uniformly terminating (see, e.g., (Bove and Capretta 2005)). Programs which truly might fail to terminate are not allowed, and finite failure is usually not considered. Finite failure simplifies code on inputs outside the intended domain, and is supported by all practical programming languages. Another approach views potentially non-terminating computations as elements of a co-inductive type, and combines such computations monadically (Capretta 2005). This method accommodates general computations indirectly, and requires co-inductive types. In contrast, the present work provides direct support for general computations, and does not rely on co-inductive types.

Extensional Type Theory. In (Constable and Smith 1987) and consequent literature, the NUPRL type theory is extended to accommodate partial functions via liftings $\bar{A}$ of total types A. Possibly diverging terms may inhabit $\bar{A}$. Since NUPRL has an undecidable type checking problem, the technical problems encountered are different than for intensional theories. Previous work adding lifted types to the Calculus of Constructions sacrificed decidability of type checking (Audebaud 1991). The recently proposed Observational Type Theory (`OTT`) supports extensionality while retaining decidable type checking (Altenkirch et al. 2007). `OTT` cannot directly accommodate truly non-terminating functions, however, and has not yet been extended with co-inductive types.

Logics of Partial Terms. Logics of partial terms support reasoning about termination and non-termination of partial recursive functions (Stärk 1998; Beeson 1985). These systems are not based on the Curry-Howard isomorphism and hence do not suffer from the problems associated with supporting partial functions in type theory. On the other hand, they lack the expressiveness and conceptual unity of type theory for writing and verifying typed programs.

Dependent Types for Programming. EPIGRAM is a total type theory proposed for practical programming with dependent types (McBride and McKinna 2004). 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). Hoare Type Theory supports internal verification of possibly non-terminating, imperative programs, but at present does not support external verification of such programs (Nanevski and Morrisett 2005).

DFOL. A final piece of related work develops a dependently typed first-order logic (DFOL) (Rabe 2006). The logic of OPTT improves upon this system by allowing both term constructors and computational functions to accept proofs as arguments, while remaining first-order (in the sense of normalization complexity).

3. Terms and Types

The syntax for OPTT terms and types is given in Figure 1. The syntax-directed classification rules for terms are given in Figure 2, with those for types and kinds omitted for space reasons. All classification rules in this paper compute a classifier as output for a given context Γ and expression as input. The rules operate modulo definitional equality, defined below. The syntax-directed nature of these classification rules, together with the proof rules presented subsequently, imply decidability of classification. A few points are needed before turning to an overview of the constructs.

Meta-variables. We write P for proofs, and F for formulas, defined in Section 4. 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 3.2).

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 syntactically is to keep definitional equality from depending on typing. A simple static *specificationality analysis*, not described here for space reasons, ensures that specificational inputs are used only in specificational arguments. This is performed as a separate analysis from typing, and so the typing rules ignore specificationality annotations. In the GURU implementation, the programmer explicitly marks function inputs as specificational, while the specificationality annotations on arguments are inferred during type checking.

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 possible 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 variables, constructors, and applications of constructors to terminating terms. Types and proofs are also terminating. We omit the simple rules for the `Terminates` judgment. The class of terms which `Terminates` recognizes as terminating is expanded in Section 6 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). 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.

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

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

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

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

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

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

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

$$
\frac{\begin{array}{l}\Gamma \vdash t : \langle d\; \bar{X} \rangle \quad \Gamma \vdash T : \texttt{type} \\ \forall\; i \leq n. \\ (\Gamma, \Delta_i, x : \{t \texttt{=} (c_i\; \bar{x}_i)\}, y : \{\langle d\; \bar{X} \rangle \texttt{=} T_i\} \vdash s_i : T)\end{array}}{\begin{array}{l}\Gamma \vdash \texttt{match}\; t\; \texttt{by}\; x\; y\; \texttt{with} \\ \quad c_1\; \bar{s}_1\; \bar{x}_1 \texttt{=>} s_1 | \ldots | c_n\; \bar{s}_n\; \bar{x}_n \texttt{=>} s_n\; \texttt{end} \;:\; T\end{array}}
$$

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

Figure 2. Term Classification

3.1 Overview of Constructs

Our `cast`-terms witness to the type checker that a term may be viewed as having an equal type. For the benefit of `casts`, `match`-terms bind variables for assumptions of equalities in the cases. Specifically, assumption variables x and y are bound (just) in the bodies of the cases, and serve as assumptions of different equalities in each case: the former that the scrutinee equals the pattern, and the latter that the scrutinee's type equals the pattern's type. The same assumption variables (x and y) are assigned different classifiers in the different cases.

In `fun`-abstractions, the bound variable x immediately following the `fun` keyword can be used for recursive calls in the body of the abstraction. It may be omitted, in which case the type annotation following the `fun`-term's declarations can also be omitted. Recursive multi-arity `fun`-terms as described in Figure 1 cannot always be translated into nested unary `fun`-terms, due to the dependent typing. The `abort` term cancels all pending evaluation. It is annotated with a type to facilitate type computation. A related construct, not otherwise mentioned, is `impossible` $P\; T$. This is definitionally equal to an `abort`, but documents via a proof P of a contradiction that execution cannot reach this point.

Our `let`-terms are as usual, except that like `match`-terms, they also bind an assumption variable. In a `let`-term, the variable y, bound in the body of the `let`-term, serves as an assumption of the equality $x = t$. GURU includes a mechanism for local macro definitions at the term, type, formula, and proof levels, not presented here for space reasons.

Finally, to make use of proved existentials in code, there is a term-level existential elimination, `existse_term`. This is similar to proof-level existential elimination (see Section 4), but it passes the element proven to exist to a term-level function as a specificational argument. Marking the argument as specificational ensures that computation cannot depend on the value of this element.

3.2 Definitional Equality

Proofs, type annotations, and specificational data are of interest only for type checking, and are dropped during evaluation. Our 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 3.5 below).
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. Notice that types are dropped only where used

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

Figure 3. Dropping Annotations

in terms. So $|T|$ is not ! for any type T. Note that dropping annotations is defined on both typeful and type-free 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-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 disequality to work more crucially when we consider functional modeling in Section 7 below.

3.3 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, for the benefit of proof rules below, 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.

3.4 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. Impossible cases are detected by match failure. Systems with richer definitional equality still need casts in general, at which point they can become problematic for external reasoning.

$$
\begin{array}{l}
\texttt{Define}\ c : A := G. \\
\\
\texttt{Inductive}\ d : K := c_1 : D_1 \mid \ldots \mid c_k : D_k\,. \\
\\
\textit{where} \\
\quad D \ ::= \ \texttt{Fun}(\bar{y} : \bar{A}).\langle d\ Y_1\ \ldots\ Y_n\rangle \\
\\
\quad K \ ::= \ \texttt{type} \mid\mid \texttt{Fun}(x : B).\,K \\
\\
\quad B \ ::= \ \texttt{type} \mid\mid T
\end{array}
$$

Figure 4. Commands

3.5 Commands and Datatypes

We avoid the uninformative formalization of a typing signature declaring and defining constants, and instead adopt an implementor's perspective on declarations and definitions. Input to an implementation of OPTT is a sequence of commands, with syntax described in Figure 4. Here, G ranges over terms, types and type families, formulas, and proofs (defined in Section 3 above and 4 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. The syntax also prohibits type constructors from accepting proofs as arguments.

4. Proofs and Formulas

The syntax of OPTT formulas and selected proof constructs 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 exclude it for simplicity. The syntax-directed classification rules are given in Figures 6, 7, and 8. Classification of proofs and formulas, as of terms and types, is easily seen to be decidable. Note that proofs may contain term- and type-level variables, so the first rule of Figure 7 is indeed needed. There are several additional points to mention:

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. In addition to the equational principles mentioned here, there are injectivity principles for term and type constructors, omitted for space reasons. In Section 11, we consider several additional equational principles.

Quantified formulas. The first rule for classifying quantified formulas disallows the body of such a formula from depending on the bound variable, if the bound variable ranges over proofs of formulas. This just enforces that implications are not dependently typed, and similarly for conjunctions, even though they are written for compactness using `Forall` and `Exists`.

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

Contexts. All contexts must have at least one occurrence of the primary hole $*$. Insertion of an expression into a hole is capture-avoiding. This disallows truly extensional reasoning about `fun`-terms and Fun-types (discussed further in Section 11 below). The context $\hat{F}$ used in the syntax for `existsi` is just a formula with a hole indicated by $*$. Contexts Y^{+} for equational congruence

$$
\begin{array}{rcl}
F & ::= & \mathit{Quant}(x\ :\ A).\ F \mid\mid \{Y_1 \stackrel{?}{=} Y_2\} \\
\\
\mathit{Quant} & \in & \{\texttt{Forall}, \texttt{Exists}\} \\
\stackrel{?}{=} & \in & \{\texttt{=}, \texttt{!=}\} \\
\\
P & ::= & x \mid\mid \texttt{!} \mid\mid \texttt{foralli}(x\ :\ A).\ P \mid\mid [P\ X_1 \cdots X_n] \mid\mid \\
& & \texttt{existsi}\ t\ \hat{F}\ P \mid\mid \texttt{existse}\ P\ P' \mid\mid \\
& & \texttt{join}_{n,m}\ t\ t' \mid\mid \texttt{symm}\ P \mid\mid \texttt{trans}\ P\ P' \mid\mid \\
& & \texttt{cong}\ Y^+\ \bar{P} \mid\mid \texttt{contra}\ P\ F \mid\mid \cdots \mid\mid \\
& & \texttt{induction}(\bar{x}\ :\ \bar{A}) \\
& & \quad \texttt{by}\ x\ y\ z\ \texttt{return}\ F\ \texttt{with} \\
& & \quad c_1\ \bar{x}_1\ \texttt{=>}\ P_1 \mid \cdots \mid c_n\ \bar{x}_n\ \texttt{=>}\ P_n\ \texttt{end}
\end{array}
$$

Figure 5. Formulas (F) and Proofs (P)

$$\frac{\Gamma \vdash F_1 : \texttt{formula} \quad \Gamma \vdash F_2 : \texttt{formula}}{\Gamma \vdash \mathit{Quant}(x\ :\ F_1).\ F_2 : \texttt{formula}}$$

$$\frac{\Gamma \vdash T : \texttt{type} \quad \Gamma, x\ :\ T \vdash F : \texttt{formula}}{\Gamma \vdash \mathit{Quant}(x\ :\ B).\ F : \texttt{formula}}$$

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

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

Figure 6. Formula Classification

reasoning (`cong`) allow replacements to take place everywhere in terms and types. At the term level, this is more liberal than replacement using the evaluation contexts of the operational semantics. The more liberal policy is very convenient when writing proofs in OPTT, since it allows replacement of arguments in function calls, which would not be directly allowed using contexts E. Despite this more liberal policy, equality is intensional (see Section 9.2).

`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.

Induction. Classification for `induction`-proofs is not stated in the figure, for space reasons. These proofs are similar to a combination of recursive `fun`-terms and `match`-terms. A third assumption variable is bound in the cases, for the induction hypothesis. The last classifier in the list $\bar{B}$ (see Figure 5) 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 $\texttt{Forall}(\bar{x}\ :\ \bar{A}).F$.

4.1 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

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash A : \mathit{sort}(A) \quad \Gamma, x\ :\ A \vdash P : F}{\Gamma \vdash \texttt{foralli}(x\ :\ A).\ P : \texttt{Forall}(x\ :\ A).\ F}$$

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

$$\frac{\Gamma \vdash P : \hat{F}[X] \quad \Gamma \vdash X : A \quad \texttt{Terminates}\ X}{\Gamma \vdash \texttt{existsi}\ X\ \hat{F}\ P : \texttt{Exists}(x : A).\hat{F}[x]}$$

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

$$\frac{\Gamma \vdash P : \{Y\ \texttt{!=}\ Y\} \quad \Gamma \vdash F : \texttt{formula}}{\Gamma \vdash \texttt{contra}\ P\ F : F}$$

Figure 7. Logical Inferences

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

$$\frac{\Gamma \vdash P : \{Y \stackrel{?}{=} Y'\}}{\Gamma \vdash \texttt{symm}\ P : \{Y' \stackrel{?}{=} Y\}}$$

$$\frac{\Gamma \vdash P_1 : \{Y_1 = Y_2\} \quad \Gamma \vdash P_2 : \{Y_2 \stackrel{?}{=} Y_3\}}{\Gamma \vdash \texttt{trans}\ P_1\ P_2 : \{Y_1 \stackrel{?}{=} Y_3\}}$$

$$\frac{\Gamma \vdash P : \{Y = Y'\}}{\Gamma \vdash \texttt{cong}\ Y^+\ P : \{Y^+[Y] = Y^+[Y']\}}$$

Figure 8. Example Equational Inferences

a term to a normal form:

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

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

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

Most frequently used in practice is `join`, for joining terms at a normal form:

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

The most sophisticated tactic, developed in Adam Petcher's Master's thesis, is `hypjoin` (Petcher 2008). This is like `join`, but operates modulo a set of proved (possibly non-normalized) ground equations, given as inputs to the tactic by the user. The tactic essentially implements congruence closure modulo evaluation.

All these tactics 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.

5. Simple Examples

We consider here three 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 and third cannot, for different reasons, be done in type theories like Coq's. A larger case study is discussed in a subsequent section.

5.1 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 expresssing some form of proof irrelevance (Hofmann and Streicher 1998; The Coq Development Team 2004). 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` uses casts in several places. For example, the body of `append` is as follows, where `P1` and `P2` are short equational proofs omitted here:

```
match l1 by u v with
  vecn A' => cast l2 by P1
| vecc A' n' x l1' =>
  cast
    (vecc A' (plus n' m) x
      (vec_append A' n' m l1' l2))
  by P2
```

The issue is that `l2` has type `<vec A m>`, but the expected return type of the function is `<vec A (plus n m)>`. So in each case, a cast is used with a proof `P1` or `P2`, which shows that `{ 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 OPTT, 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`.

5.2 Lists with Iterated Difference

In the previous example, we prove terms equal which have provably equal but not definitionally equal types. Such proofs can be done in type theories like Coq's. We may push this example further in OPTT, however, to prove terms equal with types which are not even provably equal. Consider the following list type, where the type accumulates not the length of the list, but the iterated difference of the list values:

```
Inductive mvec : Fun(n:nat).type :=
  mvecn : <mvec Z>
| mvecc : Fun(spec n:nat)(x:nat)(l:<mvec n>).
            <mvec (minus x n)>.
```

We may define `append` on such lists as before (although here we do not bother to specify the relationship between the index of the output list and that of the input lists).

```
Inductive append_t : type :=
  mk_append_t: Fun(spec n:nat)(l:<mvec n>).append_t.

fun append(spec n m:nat)
          (l1 : <mvec n>)(l2 : <mvec m>):
          append_t.
  ...
```

We may again prove associativity of `append`:

```
 Forall(n1 : nat)(l1 : <mvec n1>)
       (n2 n3 : nat)(l2 : <mvec n2>)(l3 : <mvec n3>)
       (n12 n23:nat)(l12:<mvec n12>)(l23:<mvec n23>)
       (u1:{ (append l1 l2) = (mk_append_t l12)})
       (u2:{ (append l2 l3) = (mk_append_t l23)}).
   { (append l12 l3) = (append l1 l23) }
```

The critical difference here is that, due to the non-associativity of iterated difference, the type indices in question are not provably equal. In fact, we may easily construct examples where they are provably disequal. For example, take the singleton lists containing 3, 2, and 1, respectively: the two orders of appending give rise to indices 0 and 2. OPTT allows provably equal terms with disequal types, because equality in OPTT is operational equality on type-free terms. Provable equality in type theories like Coq's are not operational in this sense, and theorems like associativity of append on lists with iterated difference cannot be proved.

5.3 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. For

substitution on open terms, see the case study below (Section 12). 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:

```
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). 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, stated in Section 11 below, for reasoning by induction on the structure of a computation which is (assumed here to be) terminating, namely `(lterm_eval e1)`.

6. Termination Casts

Universal elimination and existential introduction require terminating terms, up until now taken to be just constructor terms. 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 $(f\ \bar{x}) = r$. Now the original universal instantiation can be done with r, translating between r and $(f\ \bar{x})$ 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 a equal to a constructor term, or else proves totality of the head (call it f) of t, in the sense mentioned above. The basic design of OPTT makes this addition straightforward, since termination casts are computationally irrelevant. We extend our definitional equalities by dropping `terminates`-annotations:

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

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

7. Functional Modeling, Ownership Annotations

Inspired by a suggestion of Swierstra and Altenkirch, we sketch an extension of OPTT to support 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. Functions marked specificational must be replaced during compilation.

We extend OPTT with ownership annotations `unique` and `unique_owned`, qualifying function inputs similarly to `spec`. Inputs marked `unique` must be consumed by the function exactly once. Those marked `unique` may be used but must not be consumed. 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.

OPTT's distinction 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. So computational functions may not violate the abstraction boundary imposed 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. Examples of this approach are described below (Section 12).

8. 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 not`dec` 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. Reclaimed data items are placed on per-constructor free lists. When one of these is used to satisfy a new allocation request, its subdata are put on their free lists. We thus get an incremental memory reclamation with much more constrained worst-case running time than garbage collection. 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.

9. Cut Elimination

In this section we sketch consistency of the logic via analysis of canonical forms of closed proofs of atomic formulas, obtained by a two-step normalization process. First, standard logical cuts are removed, then equational cuts.

9.1 Logical Cut Elimination

Because OPTT's logic has been designed to have a low proof-theoretic strength, a textbook proof of strong normalization for reduction of logical cuts can be applied with only a few minor adaptations.

THEOREM 1 (Logical Cut Elimination). *Logical cuts can be removed by reduction in a finite number of steps from any* OPTT *proof.*

Next, we must establish type safety for reductions of logical cuts. Type preservation is easily established, thanks to proof irrelevance.

THEOREM 2 (Type Preservation for Logical Reduction). *If* $\Gamma \vdash P : F$ *and* $P \rightsquigarrow P'$, *then* $\Gamma \vdash P' : F$.

The second part of type safety is progress. A complication arises here, due to the possible presence of casts in constructor terms. We assume an obvious notion of being stuck.

THEOREM 3. *If* $\vdash P : F$, *then* P *is not stuck.*

Cast Shifting. We must show that P cannot get stuck because an `induction`-proof analyzing datatype $\langle d\ \bar{Y}\rangle$ is applied to a term containing some casts. Briefly, this is done by showing that we may always shift such casts out of the way, using an approach similar to that used by Chapman in a formalization of a Martin-Löf type theory (Chapman 2008). Casts between types of different structure (such as a function of 3 arguments to a function of 2) must contain a proof of an equation that contradicts our equational theory (where such types are provably disequal). The entire subproof can then be eliminated by the consequent contradiction.

9.2 Equational Cut Elimination

Every equation $\{t = t'\}$ or $\{T = T'\}$ provable in the empty context satisfies one of the following properties (modulo definitional equality):

1. t and t' are both diverging.

2. t and t' are joinable at an inactive or stuck term, the latter considered because we may reason about ill-typed terms.

3. T and T' are of the forms $R[\bar{t}]$ and $R[\bar{t}']$ for some type expression R with holes in any non-binding position, and lists of terms $\bar{t}$ and $\bar{t}'$, with corresponding elements provably equal.

A related characterization for disequations is omitted here.

THEOREM 4 (Consistency). *There are atomic formulas which are not provable in the empty context.*

THEOREM 5. *Types provably equal in the empty context have the same type structure.*

10. Type Soundness

We state type soundness using a typeful version of the operational semantics, which operates on terms with all their type and proof annotations. This typeful evaluation must insert casts as it goes to preserve typing. Consider evaluation of a term $E[R]$, with E an evaluation context, which happens to be of type $T[R]$, with R a redex. Suppose we contract R to t, causing $E[R]$ to reduce to t'. Then the type becomes $T[t]$. If the hole in T is immediately inside a type inside a term, there is no difficulty: $T[t]$ is definitionally equal to $T[R]$, since types inside terms are dropped by definitional equality. If the hole in T does not occur inside a type inside a term, then $E[t]$ might not be definitionally equal to $E[R]$. So typeful evaluation must insert a cast around t', using a proof that $\{T[t] = T[R]\}$. Type-preserving evaluation must also shift casts off of applied functional terms and off terms scrutinized by `match`. The shifting technique described above works, unless the cast uses a proof of an equation violating type structure. But this cannot happen in the empty context, thanks to Theorem 5.

THEOREM 6 (Preservation). *If* $t : T$ *and* $|t| \rightsquigarrow t'$ *then there exists* t'' *with* $|t''| = t'$ *such that* $t'' : T$.

THEOREM 7 (Progress). *If* $t : T$, *then* t *is not stuck.*

11. Equational Extensions

In OPTT, as in OTT, extension of the equational theory is much easier than in traditional intensional type theories (Altenkirch et al. 2007). Here, we see how to extend OPTT with new equational reasoning principles. For each extension, we must verify that it cannot violate the property stated for (unextended) OPTT in Theorem 5, since type soundness depends upon this.

Finite Failure Clash. The following simple principle can be easily added:

$$\frac{}{\Gamma \vdash \texttt{aclash}\ I : \{\texttt{abort ! != } I\}}$$

Parametric Extensionality. We may extend the theory with a limited form of extensionality for `fun`-terms using the following rule:

$$\frac{\Gamma \vdash P : \{t \stackrel{?}{=} t'\}}{\Gamma \vdash \texttt{pext}\ r(\bar{x} : \bar{A}).P : \{\texttt{fun}\ r(\bar{x} : \bar{A}).t \stackrel{?}{=} \texttt{fun}\ r(\bar{x} : \bar{A}).t'\}}$$

This principle allows us to equate abstractions by equating their bodies, but only using equational inferences, and critically, not induction. This is because the context is not extended, so only (untyped) equational reasoning is possible.

With full extensionality, the context is extended. This principle is not compatible with OPTT, for with it, we could prove that $\texttt{fun}(x : \textit{empty}).0$ is equal to $\texttt{fun}(x : \textit{empty}).1$. This is because we would have an assumption that *empty* is inhabited. But the equational theory of OPTT does not require type correctness of applications. Hence, these `fun`-terms could be applied to terms (which do not have type *empty* in the empty context), and then reduced using `join`. We would easily derive $\{0 = 1\}$ this way. A more typeful theory of equality could solve this problem, but carries its own costs.

Computational Inversion. We may deduce that a term t terminates from a proof that $E[t]$ terminates (where E is an evaluation context). In the empty context, this principle is valid under our characterization of provable equality.

$$\frac{\Gamma \vdash P : \{E[t] = I\} \quad \Gamma \vdash t : T}{\Gamma \vdash \texttt{cinv}\ t\ P : \texttt{Exists}(x : T).\{t = x\}}$$

Computational Induction. Stronger than computational inversion is a principle of computational induction, indirectly supported as follows. Suppose defined a standard inductive datatype for unary natural numbers. For any term t of the form $\texttt{fun}\ r(\bar{x} : \bar{A}) : T.t'$, let t^{nat} be the following term:

$$\begin{array}{l}\texttt{fun}\ r(n : nat)(\bar{x} : \bar{A}) : T. \\ \quad \texttt{match}\ n\ \texttt{by}\ x\ y\ \texttt{with} \\ \quad\quad \texttt{Z => abort !} \\ \quad\quad \mid \texttt{S n' =>}\ [(r\ n')/r]t' \\ \quad \texttt{end}\end{array}$$

This is just like t, except that it takes a *nat* n to use as computational budget in a standard way: when the budget is out, t^{nat} aborts; otherwise it behaves like t.

Suppose for simplicity that all functions used in the body t', other than r, are provably terminating. Then termination (without aborting) of t^{nat} on some inputs provably entails termination of t on those inputs, in unextended OPTT. The converse, however, does not appear to be provable. Provable equivalence of termination for t and t^{nat} would allow reasoning by induction on the structure of a terminating computation of t indirectly, by reasoning by induction on the computational budget n for a terminating computation of $(t^{nat}\ n)$. To obtain provable equivalence, we may extend OPTT with the following principle, which is already valid in the empty context under our characterization of provable equality:

$$\frac{\Gamma \vdash P : \{(t\ \bar{X}) = I\}}{\Gamma \vdash \texttt{cind}\ P : \texttt{Exists}(n : nat).\{(t^{nat}\ n\ \bar{X}) = I\}}$$

12. Case Study: Incremental LF

This section describes a larger case study carried out in OPTT, 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 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 solver's results. Efficienct 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*.

12.1 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.

12.2 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 usual 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.

12.3 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) }
```

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 7, 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.

12.4 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_nexts`. 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 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. The separation of terms and proofs in OPTT provides a foundation for an easy fix to this problem. 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 avoid distracting problems such as specifying a sensible behavior of the `size` construct on proofs, types, and formulas.

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).

12.5 Specifications and Proofs

The central routine, called `check`, to perform LF type checking on textual input has the following type, which we will consider in detail. The notation `@<...>` is for formula-level application of defined predicates (not mentioned previously), easily accomodated in definitional equality.

```
Fun(unique pb_stdin:pb_stdin_t)
   (unique symbols:symbols_t)
   (nextid:var)
   (spec symok:@<symbols_ok nextid symbols>)
   (create:bool)
   (expected:<option trm>)
   (bndexpected:{(bndopttrm nextid expected) = tt})
   (owned where:string).
  unique <check_t nextid symbols create expected>
```

This type says that `check` consumes a `pb_stdin` ("pushback standard input"), which supports reading the next character from stdin and pushing characters back; a symbol table; the next identifier to use (`var` is defined to be `word`, for 32-bit words, as discussed above); a proof of an invariant relating the symbol table and `nextid`, the latter as an upper bound on symbols in the symbol table; a flag `create` controlling whether we are in creating or non-creating mode; an optional `expected` expression, which is supplied if we are in checking mode and omitted in synthesizing mode; a proof that if the optional expression is present, all variables in it are strictly bounded above by `nextid`; and a string to be printed in any error messages generated. Given these inputs, `check` produces a unique element of the type `<check_t nextid symbols create expected>`, defined with a single constructor as follows:

```
Inductive check_t
  : Fun(nextid:var)(symbols:symbols_t)
       (create:bool)(expected:<option trm>).type :=
  mk_check
  : Fun(spec nextid:var)(spec symbols:symbols_t)
       (spec create:bool)(spec expected:<option trm>)
       (unique pb_stdin:pb_stdin_t)
       (unique symbols':symbols_t)
       (nextid':var)
       (spec T:trm)
```

```
      (k:<tcheck_t nextid' symbols create T>)
      (K:<Tcheck_t nextid' expected T>)
      (nle : { (vle nextid nextid') = tt })
      (U : { (trie_interp symbols) =
             (trie_interp symbols')}).
    <check_t nextid symbols create expected>.
```

This `mk_check` constructor first takes four specificational arguments needed to describe relationships between the computational outputs of `check` and its inputs. It then packages up these outputs: the next `pb_stdin` to use (since the one input to `check` is consumed by reading from it); an updated symbol table `symbols'`; the next identifier to use (`nextid'`); two data elements `k` and `K`, to be discussed next, related via a specificational type `T`; a proof (`nle`) that the original `nextid` is less than or equal to the new `nextid'`; and a proof that the updated symbol table has the same interpretation, as a list of (key,value) pairs, as the starting one. Note that the symbol tables in general will not be equal, even though their interpretations are, since inserting and then removing an element into the table may result in a table with a different form (due to the overlap between `trie_exact` and `trie_next`, mentioned above).

The data element `k` packages up a term and a specificational declarative LF typing derivation for it if we are in creating mode; and otherwise packages up nothing computational. The appropriate value for the `create` flag is used as an index in each case, to enable static checking of consistent mode usage:

```
Inductive tcheck_t
  : Fun(nextid:var)(symbols:symbols_t)
       (create:bool)(T:trm).type :=
  tcheck_ff
  : Fun(spec nextid:var)
       (spec symbols:symbols_t)
       (spec T:trm).
      <tcheck_t nextid symbols ff T>
| tcheck_tt
  : Fun(spec nextid:var)
       (spec symbols:symbols_t)
       (spec T:trm)
       (t:trm)
       (spec d:<deriv (gs_ctxt symbols) t T>)
       (bt : { (bndtrm nextid t) = tt}).
     <tcheck_t nextid symbols tt T>.
```

The type `<deriv (gs_ctxt symbols) t T>` in the second case is inhabited by encoded proofs in a declarative presentation of LF, to show that `t` has type `T` in context `(gs_ctxt symbols)`. This function `gs_ctxt` is based on the `trie_interp` function mentioned above: it maps a symbol table to a typing context, discarding the strings in the symbol table.

The final piece of the output of `check`, the item `K`, packages up either a synthesized type (if in synthesizing mode), or else a proof of α-equivalence of the expected type and the type `T` which the created term (if any) has. The appropriate member of the `option` type indexes `Tcheck_t`, again to enable statically verifying consistent mode usage.

```
Inductive Tcheck_t
 : Fun(nextid:var)
      (expected:<option trm>)
      (T:trm).type :=
  Tcheck_nothing
  : Fun(spec nextid:var)
       (T:trm)
       (bT:{(bndtrm nextid T) = tt}).
     <Tcheck_t nextid (nothing trm) T>
| Tcheck_something
```

benchmark	size (MB)	C++ impl	GOLFSOCK	TWELF
cnt01e	2.6	1.3	2.0	14.0
tree-exa2-10	3.1	1.7	2.5	18.6
cnt01re	4.6	2.4	3.6	218.4
toilet_02_01.2	11	5.8	8.8	1143.8
1qbf-160cl.0	20	10.0	14.1	timeout
tree-exa2-15	37	19.9	31.2	timeout
toilet_02_01.3	110	58.6	89.7	exception

Figure 9. Checking times in seconds for QBF benchmarks

```
: Fun(spec nextid:var)
     (spec T eT:trm)
     (spec nextid':var)
     (nle:{(vle nextid nextid') = tt})
     (u : { (acanon nextid' T) =
            (acanon nextid' eT)}).
   <Tcheck_t nextid (something trm eT) T>.
```

The function `acanon` is for putting terms in α-canonical form. Allowing α-equivalence from a `nextid'` greater than or equal to the current `nextid` facilitates resetting the next id when we are in non-creating/checking mode.

12.6 Statistics

The code for the `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.

12.7 Empirical Results

Figure 9 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. The results show GOLFSOCK is around 50% 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.

13. Conclusion

Operational Type Theory combines a dependently typed programming language with a first-order theory of its untyped evaluation.

By separating proofs and programs, contrary to the Curry-Howard isomorphism, we free programs to include constructs like general recursion which are problematic for proofs; and provide a principled basis for proofs to reason about programs with type annotations dropped, via untyped operational equality. The robustness of OPTT's central design ideas is shown by its ability to accomodate extensions like termination casts, as well as uniqueness and ownership annotations. Functional modeling based on the latter enables verification of non-functional code. The case study and empirical evaluation of the incremental LF checker GOLFSOCK, written in GURU, demonstrates that the OPTT methodology can be applied to build efficient verified programs.

Acknowledgements: Thorsten Altenkirch for detailed 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

T. Altenkirch. Integrated verification in Type Theory. Lecture notes for a course at ESSLLI 96, Prague, 1996. Available from the author's website.

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.

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

P. Audebaud. Partial Objects in the Calculus of Constructions. In *Proceedings 6th Annual IEEE Symposium on Logic in Computer Science*, pages 86–95. 1991.

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.

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

F. Blanqui, J.-P. Jouannaud, and P.-Y. Strub. From Formal Proofs to Mathematical Proofs: a Safe, Incremental Way for Building in First-Order Decision Procedures. In *Proc. 5th IFIP Conference on Theoretical Computer Science*. Springer-Verlag, 2008.

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

E. Brady, C. McBride, and J. McKinna. Inductive Families Need Not Store Their Indices. In *TYPES*, pages 115–129, 2003.

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

J. Chapman. Type Theory Should Eat Itself. In A. Abel and C. Urban, editors, *Proc. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.

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. Constable and S. Smith. Partial Objects in Constructive Type Theory. In *Proceedings of the Symposium on Logic in Computer Science*, pages 183–193, 1987.

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. *Dependently Typed Functional Programs and Their Proofs*. PhD thesis, 1999.

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.

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.

F. Rabe. First-Order Logic with Dependent Types. In N. Shankar and U. Furbach, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *LNCS*, pages 377–391. Springer, 2006.

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

R. Stärk. Why the constant 'undefined'? Logics of partial terms for strict and non-strict functional programming languages. *J. Funct. Program.*, 8 (2):97–129, 1998.

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.

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.