Fundamenta Informaticae XX (2008) 1–24
IOS Press

A Core Operational Type Theory

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

Edwin Westbrook
CS, Rice University
ewestbro@cse.wustl.edu

Abstract. A core calculus for Operational Type Theory (OPTT) is developed. OPTT is a new type theory designed to support more general programming than directly supported in traditional total type theories based on the Curry-Howard isomorphism. 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 are dropped during formal reasoning. This greatly simplifies verification problems by reducing the need to reason about proofs and types when reasoning about programs.

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

*Partially supported by NSF grant CCF-0448275.

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. Also, there is no need to keep the logic intuitionistic: non-constructive reasoning could be permitted. The strict separation of programs and proofs ensures that computations cannot depend on proofs, and hence non-constructive proofs cannot pose problems for (constructive) computations. In practice, a need for non-constructive reasoning has not emerged in verified programming with OPTT, and so we take OPTT's logic to be intuitionistic in this paper.

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 [1]. 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 [13]. One answer is heterogeneous equality [16]. Another is contextual definitional equality [5].

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 type annotations, and proofs in explicit casts, which are used to establish type equivalences beyond definitional equality. Compilation to erased form typically drops such annotations. 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. 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 goal of the present paper is to establish in some detail the meta-theory of a core OPTT. OPTT has been implemented in a tool called GURU, including a type/proof checker and a compiler to efficient C code. Several medium-sized case studies (on the order of thousands of lines of code and proof) have been carried out. Nevertheless, it is beyond the scope of this paper to demonstrate the practical utility of OPTT. The focus here is on the meta-theory of a core calculus for OPTT, and its benefits from a theoretical and meta-theoretical point of view. For this reason, we omit consideration of numerous features implemented in GURU. Some of these are non-trivial extensions, whose formal meta-theoretic consideration must remain to future work; while others would require less significant meta-theoretic work, but distract from the study of OPTT's essential features. We begin by considering additional related work, and then proceed to the definition of the language, and its meta-theoretic properties.

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., [6]). 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 [8]. 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 [11] 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 [3]. Observational Type Theory (OTT) supports extensionality while retaining decidable type checking [2]. 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 [22, 4]. 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 [17]. Xi's ATS and a system proposed by Licata and Harper have similar aims, but allow general recursion [15, 10]. 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 [19, 21]. Hoare Type Theory supports internal verification of possibly non-terminating, imperative programs, but at present does not support external verification of such programs [18].

DFOL. A final piece of related work develops a dependently typed first-order logic (DFOL) [20]. 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).

Multi-arity notations. We write $\texttt{fun}\ x(\bar{x} : \bar{A}) : T.\ t$ for $\texttt{fun}\ x(x_1 : A_1) \cdots (x_n : A_n) : T.\ t$, with $n > 0$. 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` side condition. This is also used in quantifier proof rules below. Terminating terms here are just *inactive* terms, of the following form:

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

Inactive terms are like values as defined for the operational semantics below (Section 3.3)), except that annotations are retained: terms, types, and casts are allowed in inactive terms. While terminating terms are just the inactive terms for purposes of this paper, it is possible to expand the terminates judgment to include features like termination casts, where a proof of totality is supplied to show the type checker that a term is terminating. Such extensions are beyond the scope of this paper, however.

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.

3.1. Overview of Constructs

Our `cast`-terms witness to the type checker that a term may be viewed as having an equal type. Note that we do not require that the cast type is classifiable. This simplifies the system and is more convenient to use in practice. The cost is that $\Gamma \vdash t : T$ does not imply $\Gamma \vdash T : \texttt{type}$. Instead, we have that there is a T' provably equal to T such that $\Gamma \vdash T' : \texttt{type}$. 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

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

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

$$\frac{\Gamma(v) = A}{\Gamma \vdash v : A} \qquad \frac{\Gamma \vdash T : \texttt{type}}{\Gamma \vdash \texttt{abort}\ T : T} \qquad \frac{\Gamma \vdash T : \texttt{type} \quad \Gamma \vdash P : \texttt{False}}{\Gamma \vdash \texttt{falsee\_term}\ T\ P : T}$$

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

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

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

$$\frac{\Gamma \vdash t : \texttt{Fun}(x : A).\ T \quad \Gamma \vdash X : A}{\Gamma \vdash (t\ X) : [X/x]T}$$

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

Figure 2. Term Classification

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. These variables are in principle sufficient for casts in the cases. We omit consideration here of a term construct that witnesses that a case cannot be used due to inconsistency of the context. While such a construct is implemented in GURU, it is less critical in OPTT than in total type theories. We must anyway prove totality externally in OPTT if it is needed: there is no automatic termination checker for `fun`-terms (though of necessity there is for `induction`-proofs, discussed below). In practice, some form of automatic type refinement can greatly reduce the annotation burden on the programmer. 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. We exclude this feature from consideration here in our core OPTT.

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 considered here.

We omit consideration of a term-level existential elimination (`existse_term`), which is necessary to make use of proved existentials in code. While this is a critical feature for programming with existential proofs, it requires a consideration of computationally irrelevant (or *specificational*) data. This is because we do not wish to permit the result of computation to depend on the value of a piece of data proven to exist (and introduced into code with `existse_term`). Otherwise, we would not be justified in dropping proofs from code at runtime. While GURU implements specificational data and `existse_term`, a formal consideration of the meta-theory of this feature is beyond the scope of this paper.

We must include a term-level `False` elimination constructr (`falsee_term`), for the benefit of proofs below. This is not used in practice, but is necessary for our meta-theoretic development.

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. $Y \equiv Y^*[c]$ and $Y' \equiv Y^*[t_c]$, where c is defined non-recursively at the top level to equal t_c (see Section 3.4 below). Here and below, contexts Y^+ for are Y entities containing a hole $*$.

$\texttt{fun}\, x() : T\,.\, t$	$\Rightarrow$	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	$(\bar{x} : \bar{A}-)$	$\Rightarrow$	$(\bar{x} : \texttt{!})$
$(\texttt{spec}\, \bar{x} : \bar{A}-)$	$\Rightarrow$	$\cdot$	$\texttt{falsee\_term}\, T\, P$		$\texttt{falsee\_term ! !}$

Figure 3. Dropping Annotations

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 confluent by Newman's Lemma. We can thus define a function $|\cdot|$ to return the unique normal form of any expression under the rules. Notice that types are dropped only where used 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 using $|\cdot|$, 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 (e.g., one definitional equality for use in type checking a proof, and a different one for terms). While this turns out to be a valuable feature in practice, exploring it further is beyond the scope of this paper.

3.3. Operational Semantics

Evaluation in OpTT is call-by-value. A small-step evaluation relation is defined in a standard way in Figure 4 on terms with annotations dropped. The definition uses evaluation contexts E and values V. The latter are like values as traditionally used in operational semantics of programming languages, but allow variables. Variables can occur during partial evaluation, used below in proof checking. In the rule for applications of `fun`-terms, it is assumed that there are as many arguments ($\bar{V}$) as parameters ($\bar{x}$); and similarly in the `match` rule. Terms of the form `falsee_term ! !` cannot arise during evaluation in the empty context, as a consequence of meta-theoretic results below. They can arise during partial evaluation, however, in which case they are treated like values.

$$
\begin{array}{lcl}
E[(F\ \bar{V})] & \rightsquigarrow & E[[\bar{V}/\bar{x}, F/x]t] \\
F\ \equiv\ \texttt{fun}\ x(\bar{x} : \bar{\texttt{!}})\ :\ \texttt{!}\ .t & & \\
\\
E[\texttt{match}\ (c_i\ \bar{V})\ \texttt{by}\ x\ y\ \texttt{with}\ c_1\ \bar{x}_1\texttt{=>}s_1 | \ldots | c_n\ \bar{x}_n\texttt{=>}s_n\texttt{end}] & \rightsquigarrow & E[[\bar{V}/\bar{x}_i]s_i] \\
\\
E[\texttt{let}\ x\ =\ V\ \texttt{by}\ y\ \texttt{in}\ t] & \rightsquigarrow & E[[V/x]t] \\
\\
E[\texttt{abort !}] & \rightsquigarrow & \texttt{abort !}
\end{array}
$$

where:

$$
\begin{array}{lcl}
E & ::= & *\ ||\ (E\ X)\ ||\ (V\ E)\ ||\ \texttt{let}\ x\ =\ E\ \texttt{by}\ y\ \texttt{in}\ t\ || \\
 & & \texttt{match}\ E\ \texttt{by}\ x\ y\ \texttt{with}\ c_1\ \bar{x}_1\ \texttt{=>}\ t_1 | \ldots | c_n\ \bar{x}_n\ \texttt{=>}\ t_n\ \texttt{end} \\
V & ::= & x\ ||\ c\ ||\ \texttt{!}\ ||\ T\ ||\ P\ ||\ (c\ V_1\ \cdots\ V_n)\ ||\ \texttt{fun}\ x(\bar{x}\ :\ \texttt{!})\ :\ \texttt{!}.\ t\ ||\ \texttt{falsee\_term ! !}
\end{array}
$$

Figure 4. Small-step Evaluation

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

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 5. Commands

3.4. Datatypes

We avoid the uninformative formalization of a typing signature declaring and defining constants, and instead specify informally the kinds of datatypes that may be declared, thus adding type and term constructors to the signature. The syntax of datatype declarations is given in Figure 5. Here, K is for kinds. Datatypes may be both term- and type-indexed. We additionally restrict input types A to a (term) constructor so that if d occurs in A, it does so only if A is a type application with d as the head. More liberal forms of datatypes are of interest in practice, but for simplicity we consider just such algebraic datatypes. Also, we impose the additional restriction that input types may not contain quantification or functional abstraction over types. 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 6. For reasons that will be explained below, it is helpful to view implications as degenerate forms of universal quantifications. For symmetry, we similarly view 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 7, 8, and 9. 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 8 is indeed needed. There are several additional points to mention:

Formulas. Equations are formed between type-free terms, as well as between types. Instead of allowing any untyped terms, one could require some form of approximate typing, but this is not essential nor required in practice. For a large number of program verification examples logical disjunction (not to be confused with the boolean operation "or") is unnecessary, and indeed, GURU does not implement it. Omitting it from the formal treatment simplifies, as is well known, the strong normalization argument. Existential quantification, which also poses complications for that argument, is included, because it truly is needed for practical program verification. Implication and conjunction are viewed here as degenerate forms of universal and existential quantification, respectively. This is actually implemented in GURU, as it provides a more compact syntax for long sequences of universal variables and hypotheses. This treatment plays a more fundamental role in the present work, in the treatment of existential elimination in the cut elimination proof (Section 6).

Contexts and Holes. Holes (h in Figure 6) are numbered by n, a natural number, for the benefit of the injectivity rule. The notation $C[\bar{I}]$ is for the result of substituting the $I_1, \ldots, I_n$ for the first n holes of C. Similarly, $C[Y, \bar{Y}]$ in the injectivity rule denotes the result of substitution Y for hole $*_0$ and then $\bar{Y}$ for the next holes. For congruence, we stipulate that Y^* has at least one occurrence of the hole $*_0$, and no occurrences of other holes. The holes may occur anywhere in Y^*. This unrestricted form of congruence is needed for type preservation of term reduction (Section 8 below). Insertion of an expression into a hole is capture-avoiding. This disallows truly extensional reasoning about `fun`-terms and `Fun`-types. The classification rule for `clash` records disjointness of the range of constructors: two constructor terms are disequal if their heads are disequal, where the head of $(c\ \bar{h})$ is c and $\langle d\ \bar{h} \rangle$ is d. For purposes of this rule, we treat a constructor by itself as a 0-ary application.

Injectivity. Various injectivity rules are required. The basic `inj`-inference allows equating corresponding subterms of terms headed by the same constructor, disregarding some other, possibly different, subterms. We require those other subterms to be terminating to allow for non-trivial equation of diverging terms. This is not done in the core OPTT, but could be allowed. In such a case, unsoundness could result if the other subterms were diverging, since strictness would equate the two constructor terms on the basis of the diverging subterms. The terms equated by `inj` might not, then, be truly equal. We further stipulate that in a use of `inj`, we cannot pass from an equality on terms in the premise to an equality on types in the conclusion. This is needed for soundness, since dropping annotations from terms in our definitional equality allows, for example, (`nil bool`) and (`nil nat`) to be equated. Yet it would be unsound to conclude from this that `bool` equals `nat`.

The other injectivity rules are not, so far, needed in practice, but are required for the meta-theory. The somewhat unusual form of `dom_inj2` is chosen to avoid, at all costs, reasoning about equality of formulas. Such reasoning quickly gives rise to casts on proofs, which could not be accomodated in an

obvious way by our reducibility argument for strong normalization of proof reductions (Section 6 below). The form chosen here gives us enough information to simulate a cast on a proof at the critical point in the strong normalization proof.

Evaluation. The rule `evalstep` axiomatizes the small-step operational semantics. As in other theorem provers, higher-level tactics are needed in practice. GURU implements several of these, for joining terms with multiple steps of reduction. They are not essential to the core theory, however, and are not discussed further. Note that the use of $|\cdot|$ for obtaining the unannotated expansion of a term is used in the premise of `evalstep`, because the operational semantics is defined only for terms without annotations.

`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 $\bar{A}$ (see Figure 6) 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. Simple structural decrease at recursive call sites is checked automatically. The classifier for the proof is then of the form $\texttt{Forall}(\bar{x} : \bar{A}).F$.

`False` **elimination.** We restrict `falsee` to introduce just an atomic formula (i.e., an equation) from a contradiction. A simple meta-theoretic proof by induction on the structure of a formula F then shows that from a contradiction, we can build a proof of F using `falsee` for atoms and the corresponding introduction rule for each other form of formula. This requires `falsee_term` in the existential case. Since we can derive any formula F from a contradiction, we are justified in restricting `falsee` to atomic formulas. This, in turn, simplifies the meta-theory.

5. Advantages of Weak Definitional Equality

Before presenting the meta-theoretic results, we consider some critical aspects of OPTT's design at a high level. OPTT's definitional equality is unusual for a type theory in several respects. Most essentially, as we have seen, it drops annotations from terms. This facilitates reasoning about operational behavior, where annotations are irrelevant. But secondarily, it does not include β-reduction, or any similar computational reductions. This makes it much weaker than the conversion relations often used in type theory, which include β-reduction, and even pattern-matching and structural recursion. This is the situation, for example, in COQ. In OPTT, we cannot include the operational semantics of the language in the definitional equality, because the presence of general recursion would render testing definitional equality undecidable. Pushing conversion reasoning out of definitional equality and into propositional equality solves this problem. Taking propositional equality to operate on unannotated terms then mitigates the consequences of this, namely that we must include more casts in terms.

There is a further critical benefit of using a weak definitional equality, which is applicable even for type theories of total functions. It makes it much easier to implement other operations of the type theory so that they work modulo definitional equality. Definitional equality is supposed to be provide certain syntactic identities for free. Wherever it is used, it should abstract away completely from certain syntactic

$$
\begin{array}{rcl}
F & ::= & \mathit{Quant}(x\,:\,A).\,F \;||\; \{Y_1 \stackrel{?}{=} Y_2\} \;||\; \texttt{False} \\
\\
\mathit{Quant} & \in & \{\texttt{Forall}, \texttt{Exists}\} \\
\stackrel{?}{=} & \in & \{\texttt{=}, \texttt{!=}\} \\
\\
P & ::= & x \;||\; \texttt{!} \;||\; \texttt{foralli}(x\,:\,A).\,P \;||\; [P\;X_1 \cdots X_n] \;||\; \texttt{existsi}\;t\;F^*\;P \;||\; \texttt{existse}\;P\;P' \;||\; \\
 & & \texttt{refl}\;Y \;||\; \texttt{symm}\;P \;||\; \texttt{trans}\;P\;P' \;||\; \texttt{cong}\;Y^*\;\bar{P} \;||\; \texttt{clash}\;I_1\;I_2 \;||\; \texttt{falsee}\;P \;||\; \\
 & & \texttt{induction}(\bar{x}\,:\,\bar{A})\;\texttt{by}\;x\;y\;z\;\texttt{return}\;F\;\texttt{with}\;\;c_1\;\bar{x}_1\;\texttt{=>}\;P_1 | \cdots | c_n\;\bar{x}_n\;\texttt{=>}\;P_n\;\texttt{end} \;||\; \\
 & & \texttt{inj}\;C\;P \;||\; \texttt{dom\_inj1}\;P \;||\; \texttt{dom\_inj2}\;P \;||\; \texttt{ran\_inj}\;X\;P \;||\; \texttt{fclash}\;\texttt{Fun}(x : A).B\;\langle T\;Y\rangle \;||\; \\
 & & \texttt{evalstep}\;t\;t' \\
\\
C & ::= & (c\;\bar{h}) \;||\; \langle d\;\bar{h}\rangle \\
h & ::= & *_n
\end{array}
$$

Figure 6. Formulas (F) and Proofs (P)

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

$$
\frac{}{\Gamma \vdash \{t_1 \stackrel{?}{=} t_2\} : \texttt{formula}}
\qquad
\frac{}{\Gamma \vdash \{T_1 \stackrel{?}{=} T_2\} : \texttt{formula}}
$$

Figure 7. Formula Classification

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash P : \texttt{False}}{\Gamma \vdash \texttt{falsee}\ P\ Y_1\ Y_2 : \{Y_1 = Y_2\}}$$

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

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

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

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

Figure 8. Logical Inferences

$$\frac{I \equiv C[\bar{I}] \quad I' \equiv C'[\bar{I}'] \quad \mathit{head}(C) \not\equiv \mathit{head}(C')}{\Gamma \vdash \texttt{clash}\ I\ I' : \{I\ \texttt{!=}\ I'\}} \qquad \frac{\Gamma \vdash P : \{Y = Y'\}}{\Gamma \vdash \texttt{cong}\ Y^*\ P : \{Y^*[Y] = Y^*[Y']\}}$$

$$\frac{\Gamma \vdash P_1 : \{Y_1 = Y_2\} \quad \Gamma \vdash P_2 : \{Y_2 \stackrel{?}{=} Y_3\}}{\Gamma \vdash \texttt{trans}\ P_1\ P_2 : \{Y_1 \stackrel{?}{=} Y_3\}} \qquad \frac{\Gamma \vdash P : \{Y \stackrel{?}{=} Y'\}}{\Gamma \vdash \texttt{symm}\ P : \{Y' \stackrel{?}{=} Y\}}$$

$$\frac{\Gamma \vdash P : \{C[Y, \bar{Y}] = C[Y', \bar{Y}']\} \quad \texttt{Terminates}\ \bar{Y},\ \bar{Y}'}{\Gamma \vdash \texttt{inj}\ C\ P : \{Y = Y'\}} \qquad \frac{|t| \rightsquigarrow |t'|}{\Gamma \vdash \texttt{evalstep}\ t : \{t = t'\}}$$

$$\frac{\Gamma \vdash P : \{\texttt{Fun}(x : T).B = \texttt{Fun}(x : T').B'\}}{\Gamma \vdash \texttt{dom\_inj1}\ P : \{T = T'\}} \qquad \frac{\Gamma \vdash P : \{\texttt{Fun}(x : A).B = \texttt{Fun}(x : A').B'\}}{\Gamma \vdash \texttt{ran\_inj}\ X\ P : \{[X/x]B = [X/x]B'\}}$$

$$\frac{}{\Gamma \vdash \texttt{refl}\ Y : \{Y = Y\}} \qquad \frac{\Gamma \vdash P_1 : \{Y_1 = Y_2\} \quad \Gamma \vdash P_2 : \{Y_1\ \texttt{!=}\ Y_2\}}{\Gamma \vdash \texttt{contra}\ P_1\ P_2 : \texttt{False}}$$

$$\frac{\Gamma \vdash P : \{\texttt{Fun}(x : F^*[t, \bar{t}]).B = \texttt{Fun}(x : F^*[t', \bar{t}']).B'\} \quad *_0 \textit{ not in a term in } F^*}{\Gamma \vdash \texttt{dom\_inj2}\ F^*\ P : \{t = t'\}}$$

$$\frac{}{\Gamma \vdash \texttt{fclash}\ \texttt{Fun}(x : A).B\ \langle T\ Y\rangle : \{\texttt{Fun}(x : A).B\ \texttt{!=}\ \langle T\ Y\rangle\}}$$

Figure 9. Equational Inferences

differences in expressions, so that these differences are eliminated for purposes of other operations. Let us say that a family of judgments $\mathcal{J}$, such as typing judgments, satisfies *definitional transparency* iff whenever $\mathcal{J}[e]$ holds for some judgment $\mathcal{J}$, then $\mathcal{J}[e']$ also holds, when e and e' are definitionally equal. In a more refined fashion, we could speak about which argument positions to a judgment like typing satisfy definitional transparency.

The core typing judgments of a traditional type theory like, for example, the Calculus of Inductive Constructions (CIC) used in COQ, all satisfy definitional transparency, by design. But many extra operations are implemented in a tool like COQ beyond those core judgments. The difficulty with using a strong definitional equality is that it becomes very difficult to ensure that those other judgments also satisfy definitional transparency. This leads to a non-uniformity in the system: certain judgments are definitionally transparent, and certain others are not. This non-uniformity can lead to confusion for at least new users of such systems, and may be considered a design flaw. For example, consider the problem of applying a rewrite rule to a term modulo definitional equality. This operation is provided by the `rewrite` tactic in COQ. If definitional equality includes even just β-reduction of simply typed redexes, as it does in CIC, then applying the rule modulo definitional equality requires higher-order matching, only just recently proved decidable in general [23]. Indeed, `rewrite` does not work modulo definitional equality. In contrast, rewriting can be performed easily modulo OPTT's definitional equality, without higher-order matching, by operating on unannotated expansions of terms.

In systems like COQ, many tactics intended to aid users in the construction of proofs do not work modulo definitional equality. Rewriting tactics are a good example. Indeed, many tactics are designed to allow the user to change a goal formula into a different but definitionally equal one! Thus, the untrusted theorem proving environment, which is, as it should be by the *de Bruijn* criterion, the biggest part of the system, does not satisfy definitional transparency. It is then small consolation that the core typing judgments do. In contrast, the weakness of OPTT's definitional equality makes it possible to implement all typical theorem proving operations like rewriting or unification in a definitionally transparent way. We must just note that while the classification judgments of OPTT satisfy definitional transparency for the classifiers and contexts without qualification, for classified terms they do so only under replacement by classifiable expressions. That is, if $\Gamma \vdash e[e_1] : T$ holds for any classified expression e containing e_1, then we have $\Gamma \vdash e[e_2] : T$ if and only if $e_1 \equiv e_2$ and e_2 is classifiable in Γ. Omitting classifiability of e_2 here is not possible, since annotations are required in general for expressions to be classified, though not for expressions to be classifiers.

This critique of conversion must be qualified by one note. If definitional equality replaces all additional theorem proving operations, the lack of definitional transparency can be avoided. This sounds like a radical option, but it is being explored in the context of the Calculus of Congruent Inductive Constructions (CCIC), where definitional equality is extended to include decision procedures operating with hypotheses from the context [5]. Viewed from the perspective of definitional transparency, this impressive work can be viewed as a high-stakes gamble: if all theorem proving operations can be subsumed in definitional equality, then definitional transparency is retained. But if not, then any that remain will almost certainly be impossible to implement modulo the now highly complex definitional equality. Finally, we note that N. de Bruijn also advocates the use of a weak definitional equality, although in his case this still includes β-reduction (but not pattern matching or recursion) [7].

6. Logical Cut Elimination

Our goal is to prove consistency of OPTT's logic in a standard way, via analysis of canonical forms of closed proofs of atomic formulas. Proofs in this form are obtained via a 2-stage process. In the first stage, we eliminate *logical* cuts: that is patterns of introduction followed by elimination inferences, where the inferences in question are logical ones (Figure 8). For logical cut elimination, we prove our stage 1 reduction process is strongly normalizing (SN). The second stage of the process is considered in the next section.

By design, and as one of its main benefits, OPTT has a simple consistency proof, in both a qualitative and proof-theoretic sense, despite accomodating programs of intolerable proof-theoretic strength (i.e., diverging ones). The SN proof is based on a standard one, namely that given by Girard for Gödel's System T (which we refer to below as the PaT proof) [12]. The necessary adaptations are (1) accomodation of existentials and (2) treatment of a richer class of datatypes.

6.1. A Modified System for Existentials

It is well known that existential elimination does not fit the pattern of other elimination rules for minimal natural deduction, since the result of the elimination is an arbitrary formula unrelated to the principal formula of the elimination. For the convenient accomodation of existentials, we use a modified proof system with a different treatment of existential elimination. Replace $\texttt{existse}\ P\ P'$ everywhere in proofs with $[P'\ \texttt{existse1}P\ \texttt{existse2}\ P]$, and add these rules:

$$\frac{\Gamma \vdash P : \texttt{Exists}(x : A).F^*[x]}{\Gamma \vdash \texttt{existse1}\ P : A}$$

$$\frac{\Gamma \vdash P : \texttt{Exists}(x : A).F^*[x]}{\Gamma \vdash \texttt{existse2}\ P : F^*[\texttt{existse1}\ P]}$$

For this, we naturally must extend the syntax of our terms (t), types (T), and proofs (P) to include $\texttt{existse1}\ P$, and of proofs to include $\texttt{existse2}\ P$. Also, we extend our notion of inactive expressions I (that is, terminating expressions) and values V (from the operational semantics) to allow $\texttt{existse1}\ P$. Less trivially, we must modify our definitional equality, specifically our notion of dropping annotations. Now, we do not automatically discard all proofs. Instead, we discard them everywhere except immediately under `existse1`. There they must be retained to distinguish different entities proved to exist. We must also add an inference $\texttt{existse\_redex}\ F^*\ P$ stating that from $P : F^*[t]$, we may conclude $F^*[\texttt{existse1}\ \texttt{existsi}\ t\ F'^*\ P']$, if $P' : F'^*[t]$. This is needed for preservation of types during logical reduction. Also, in inferences $\texttt{existsi}\ X\ F^*\ P$, we forbid F^* to use its hole beneath `existse1` if X is a proof. So `existse1` terms may not depend on existential variables which range over proofs. This restriction is used in Section 6.3 below (for the definition of `Cast`).

The reader may wonder how the eigenvariable condition on `existse` in the original system is enforced in the new system. In the new system, the dependence of a conclusion on variables introduced by an existential whose proof P depends on hypotheses is tracked explicitly by $\texttt{existse1}\ P$, which is used instead of an eigenvariable. The classification rule for proof-level application (i.e., universal elimination) ensures that these dependencies are updated appropriately when the set of hypotheses supporting an `existse1`-term changes. For example, consider this proof term:

```
foralli(u:Exists(x:nat).
          Exists(v:{ (lt x 3) = tt }).
                { (lt 0 x) = tt }).
  [lt_S existse1 u 3 existse2 existse2 u]
```

Call this proof term P. Assuming that `lt_S` proves monotonicity of successor w.r.t. less-than (`lt`) (and assuming decimal notation for unary natural numbers), P proves

```
Forall(u:Exists(x:nat).
         Exists(v:{ (lt x 3) = tt }).
               { (lt 0 x) = tt }).
  { (lt (S existse1 u) 4) = tt }
```

Instantiating this universal quantifier with a P' containing free assumption variables will result in a proof term with classifier:

```
  { (lt (S existse1 P') 4) = tt }
```

The dependence of this fact on the assumptions in P' is thus tracked by the type system.

The scheme just described preserves the property that $\Gamma \vdash P : F$ implies $\Gamma \vdash F :$ `formula`, while replacing the meta-theoretically problematic existential elimination with a much more tractable variant. We omit a full soundness proof.

6.2. An Ordering on Classifiers

For the adaption of the (unary) logical relation used in the PaT proof, it is necessary to define a well-founded partial order on classifiers. First, let $<$ be a standard subterm (partial) ordering on classifiers, where as usual we include instances of quantified formulas as subterms. This includes instantiating a formula with an `existse1`-term. In OPTT, there is no quantification over formulas, so this is well-founded. Now, extend this ordering $<$ by additionally making $\langle d\ \bar{X} \rangle$ bigger than all classifiers which lack `Fun`-abstraction or quantification over `type`, and which mention just datatypes d' also mentioned in the datatype declaration for d. We must exclude abstraction and quantification over `type` in input types to term constructors, since otherwise our definition could make instances of a type like $\texttt{Fun}(A : \texttt{type}).A$ isomorphic to that type (the instance is smaller as an instance, but bigger if the `Fun`-type can be used as the type of an argument to a term constructor for the instance). Finally, take the transitive closure of the resulting relation. This yields a well-founded partial order on classifiers, sufficient for the needs of our SN proof below.

6.3. Cast Shifting

Reduction for proofs and also for terms requires cast shifting, used also in work by Chapman [9]. Cast shifting is defined by these rules, which are to be applied only if they preserve typability:

$$
\begin{array}{lcl}
(\texttt{cast}\ t\ \texttt{by}\ P\ \ t') & \Rightarrow & \texttt{cast}\ (t\ \texttt{cast}\ t'\ \texttt{by}\ \texttt{dom\_inj}\ P)\ \texttt{by}\ \texttt{ran\_inj}\ t'\ P \\
(\texttt{cast}\ t\ \texttt{by}\ P\ \ P') & \Rightarrow & \texttt{cast}\ (t\ \texttt{Cast}\ F^*\ P'\ P_1\ \cdots\ P_n)\ \texttt{by}\ \texttt{ran\_inj}\ P'\ P \\
\textit{see text for}\ \texttt{Cast}\ F^*\ P'\ P_1, \ldots, P_n. & & \\
(\texttt{cast}\ t\ \texttt{by}\ P\ \ T) & \Rightarrow & \texttt{cast}\ (t\ T)\ \texttt{by}\ \texttt{ran\_inj}\ T\ P \\
\texttt{cast}\ \texttt{cast}\ t\ \texttt{by}\ P\ \texttt{by}\ P' & \Rightarrow & \texttt{cast}\ t\ \texttt{by}\ \texttt{trans}\ P\ P'
\end{array}
$$

These rules may fail to preserve typability if the type of t (before being cast) is not a Fun-type. In that case, as stipulated, the rules are not applied. We will see that this situation cannot arise in the empty context, which is sufficient for our consistency proof. We further stipulate that after normalization with these rules, if the scrutinized term in an induction-redex is not a cast term, then a trivial cast of the form cast t by refl T, where T is the type of t, is inserted. This allows induction-redex reduction to be expressed in a uniform way. The rules are locally confluent and, since they reduce the sum of the depths of cast terms applied as functions, also terminating. Hence, this reduction relation is convergent.

We must now explain the Cast $F^*\ P'\ P_1\ \cdots\ P_n$ in the second cast shifting rule. The goal is to simulate casting the proof P', without actually introducing a construct for casting proofs. We suppose here that P proves two Fun-types equal, with domains $F^*[\bar{t}]$ and $F^*[\bar{t'}]$, respectively, for some formula context F^* whose holes do not occur in terms, and sequences of terms $\bar{t}$ and $\bar{t'}$. If P does not prove two such Fun-types equal, the reduction is not performed. Using dom_inj2, we may construct proofs P_i of $\{t_i = t'_i\}$. Figure 10 then defines the meta-level function Cast, which computes a proof of $F^*[\bar{t}]$ from the proof P' of $F^*[\bar{t'}]$, and the proofs equating the $\bar{t}$ with the $\bar{t'}$. To simplify the presentation, we assume variables come with their types, and thus dispense with a typing context. In the definition, we assume that $\bar{t}$ and $\bar{t'}$ are always such that $P_i : \{t_i = t'_i\}$. Note that we must reverse the equations via symm $\bar{P}$ in the first premise of the rule for Forall. The rather complex expression given to existsi in the Exists rule is to turn F^* into a suitable context for existential introduction, while reflecting the fact that it contains the $\bar{t}$. In the rule for types T^*, we write cong $A^*\ \bar{P}$ for a proof of $\{T^*[\bar{t}] = T^*[\bar{t'}]\}$ built transitively using cong with the $\bar{P}$ consecutively.

The algorithm maintains the invariant that in each call $\Gamma \vdash \texttt{Cast}\ A^*\ X\ \bar{P}\ =\ X'$, we have $X : A^*[\bar{t'}]$ and $X' : A^*[\bar{t}]$. This can be easily checked for the Forall rule. For the existential rule, the reasoning is more subtle. The P' we get back from the recursive call to Cast has classifier $[\texttt{existse1}\ P/x]F^*[\bar{t}]$. The difficulty here is that for our existsi inference, we need that classifier to be definitionally equal to $[X/x]F^*[\bar{t}]$. This holds, however, because if A^* is a formula, then our restriction in Section 6.1 prohibits any other existse1-term from depending on this proof existse1 P. Proofs are otherwise all definitionally equal, so the desired definitional equality holds in that case. In case A^* is a type or type, we may confirm that Cast always produces a definitionally equal expression. The rather complex transformation for the case of $\stackrel{?}{=}$-formulas happens to work for both equations and disequations.

6.4. Reduction for Proofs

Figure 11 specifies the reduction relation on proofs which we shall prove is strongly normalizing. The reduction of induction-proof redexes (specified by the first rule of Figure 11) is to be done modulo normalization of the scrutinized argument by the cast shifting relation just defined. The scrutinized argument is the one corresponding to the parameter of induction.

$$\frac{\texttt{Cast}\ A^*\ x\ \texttt{symm}\ \bar{P}\ =\ X \quad \texttt{Cast}\ [X/y]F^*\ [P\ X]\ \bar{P}\ =\ P'}{\texttt{Cast}\ \texttt{Forall}(y : A^*).F^*\ P\ \bar{P}\ =\ \texttt{foralli}(x : A^*[\bar{t}]).P'}$$

$$\frac{\texttt{Cast}A^*\texttt{existe1}\ P\ \bar{P}\ =\ X \quad \texttt{Cast}\ [\texttt{existse1}\ P/x]F^*\ \texttt{existse2}\ P\ \bar{P}\ =\ P'}{\texttt{Cast}\ \texttt{Exists}(x : A^*).F^*\ P\ \bar{P}\ =\ \texttt{existsi}\ X\ ([*_0/x]F^*)[*_0, \bar{t}]\ P'}$$

$$\texttt{Cast}\ \{Y_1^*\ \stackrel{?}{=}\ Y_2^*\}\ P\ \bar{P} = \begin{array}{l} \texttt{symm}\ \texttt{trans}\quad \texttt{cong}\ Y_2^*\ \bar{P} \\ \qquad\qquad\quad \texttt{symm}\ \texttt{trans}\ \texttt{cong}\ Y_1^*\bar{P}\ P \end{array}$$

$$\texttt{Cast}\ T^*\ t\ \bar{P}\ =\ \texttt{cast}\ t\ \texttt{by}\ \texttt{symm}\ \texttt{cong}\ T^*\ \bar{P}$$

$$\texttt{Cast}\ \texttt{type}\ T\ \bar{P}\ =\ T$$

Figure 10. Definition of `Cast`

Our reduction relation is then the compatible closure of the rules in Figure 11. A few further notes are required. First, the rather complex form of an `induction`-proof redex (determined by the first rule in the figure) is to accomodate casts on the scrutinized term. For type preservation, we must substitute proofs for the assumption variables y_1 and y_2. Second, we consider reduction only for well-classified proofs. We must, of course, prove that classifiers are preserved by reduction.

6.5. Classifier Preservation for Proof Reduction

Before proving normalization of the proof reduction defined above, we first prove classifier preservation for proof reduction. That is:

Theorem 6.1. (Type Preservation for Proof Reduction)
If $X^*[R] \rightsquigarrow X^*[C]$ for a context X^*, a redex R, and contractum C, and if $X^*[R] : A$ for a classifier A; then $X^*[C] : A$.

The presence of reductions for `existse1` necessitates the more general form of this statement: otherwise, we could phrase it in terms of proof contexts P^* and formula contexts F^*, in place of X^* and A^*, respectively. The theorem will not go through without the following extension of the system. We stipulate that definitional equality is expanded to include the equational theory determined by the reductions above. This is for the benefit of accomodating contracted proofs beneath `existse1` in the formula proved, and also of accomodating contractions of `existse1` in the type of the reduced proof. This modification may mean, a priori, that definitional equality is no longer decidable. Since this change is done solely for the benefit of this meta-theoretic argument, it is not necessary to retain a decidable definitional equality.

The proof is by induction on the structure of the context X^*. If X^* is just the context's hole, then we must just consider each redex reduction. The `existse1` and `existse2` cases are obvious. The `foralli` case follows from the following Substitution Lemma.

$$[Q\ \bar{X}\ s] \quad \leadsto \quad [\bar{X}/\bar{x}, s/x, \bar{X'}/\bar{x'}, \texttt{refl}\ (c_i\ \bar{X'})/y_1, \texttt{symm}\ P/y_2, Q/y_3]P_i$$

where:

1. $Q \equiv \texttt{induction}(\bar{x} : \bar{A})(x : d)\ \texttt{by}\ y_1\ y_2\ y_3\ \texttt{return}\ F\ \texttt{with}\ C_1 | \cdots | C_n\ \texttt{end}$
2. $s \equiv \texttt{cast}(c_i\ \bar{X'})\ \texttt{by}\ P$
3. $\forall i.(C_i \equiv c_i\ \bar{x'}\ \Rightarrow\ P_i)$

$$[\texttt{foralli}(x : A).P\ \ X] \quad \leadsto \quad [X/x]P$$

$$\texttt{existse1}\ \texttt{existsi}\ X\ F^*\ P \quad \leadsto \quad X$$

$$\texttt{existse2}\ \texttt{existsi}\ X\ F^*\ P \quad \leadsto \quad P$$

Figure 11. Reductions for Proofs

Lemma 6.1. (Substitution)
If $\Gamma, x : A_1, \Gamma' \vdash X_2 : A_2$, $\Gamma \vdash X_1 : A_1$, and $\texttt{Terminates}X_1$, then $\Gamma, [X_1/x]\Gamma' \vdash [X_1/x]X_2 : [X_1/x]A_2$; and similarly for $\Gamma, x : A_1, \Gamma' \vdash F : \texttt{formula}$.

The proof is by induction on the structure of the derivation of the first assumed classification. The requirement of termination is needed just for cases like proof-level application. Without this requirement, we might be substituting a term not judged terminating by `Terminates` for a variable (which is judged terminating by `Terminates`). That would cause the substitution instance to be untypable. Since both values and terms introduced by `existsi` are terminating terms, this restriction to terminating terms still enables the Substitution Lemma to be applied in the cases where it is needed, namely here and for type preservation for term reduction.

To return to the proof of Theorem 6.1: the case for `induction`-proofs also follows from the Substitution Lemma. The restriction imposed by the classification rule for proof-level application ensures that $\bar{X}$ and $(c_i\ \bar{X'})$ are terminating, so Substitution can be applied. Note that $\texttt{refl}\ (c_i\ \bar{X'})$ does prove, modulo definitional equality, that s (the instance of x) equals $(c_i\ \bar{X'})$ (the instance of the pattern). Similarly, $\texttt{symm}\ P$ proves that the type of the instance of x equals the type of the instance of the pattern.

This concludes the base case of the induction on the form of the context X^*. We must now prove the step case. The only problematic cases are when a reduction happens in an argument to a proof-level application, and when a reduction happens in a proof given to `existse2`. For the first case, we have P_1 applied to $P_2^*[R]$, which reduces to P_1 applied to $P_2^*[C]$. Thanks to our extension of definitional equality to include proof reductions, the types of the arguments are still definitionally equal, so the application is still typable with a definitionally equal type. Similarly, the type of the reduced `existse2`-proof is also definitionally equal to the original.

6.6. Strong Normalization

We may now adapt the PaT SN proof for OPTT's proof reduction. First, we stipulate that an `induction`-proof starting with $\texttt{induction}(\bar{x} : \bar{A})$ proves a universal of the form $\texttt{Forall}((\bar{x} : \bar{A}))$, where the double parentheses are special notation indicating that the given arguments must be supplied simultaneously. Any partial application of an `induction`-proof or its third assumption variable (for the induction hypothesis) can be modified to be type correct with respect to this new typing of `induction`-proofs, by η-expansion. So we assume this done, and hence all uses of `induction`-proofs are fully applied to their arguments. So we are considering `induction`-redexes only, and need not define reducibility for the types of `induction`-proofs. Similarly, we insist that all term constructors are fully applied. Following PaT, we further assume each variable is labeled with exactly one type, and we dispense with a typing context. The set of reducible expressions X of classifier A with data bound n, denoted RED^n_A, is then defined by recursion on norm (A, n) with respect to the lexicographic combination of the above defined classifier ordering (for A) and the usual natural number ordering (for natural number n), as follows:

1. Call A atomic iff it is an equation, `False`, or `type`. Then for atomic A, $X \in \mathit{RED}^n_A$ iff X is strongly normalizing.

2. If A is a type T, then $X \in \mathit{RED}^{(S\ n)}_A$ iff X is strongly normalizing and whenever $(c\ \bar{X})$ is a normal form of X for term constructor c, then for each X_i we have $X_i \in \mathit{RED}^n_{A'}$, where X_i has classifier A'.

3. If $A \equiv \texttt{Exists}(x : A').F$, then $X \in \mathit{RED}^n_A$ iff there exists n' such that $\texttt{existse1}\ X \in \mathit{RED}^{n'}_{A'}$ and $\texttt{existse2}\ X \in \mathit{RED}^{n'}_{F'}$, where $F' \equiv [\texttt{existse1}\ X/x]F$.

4. If $A \equiv \texttt{Forall}(x : A').F$, then $X \in \mathit{RED}^n_A$ iff for all n', and for all $X' \in \mathit{RED}^{n'}_{A'}$, there exists n'' with $(X\ X') \in \mathit{RED}^{n''}_{F'}$, where $F' \equiv [X'/x]F$.

The data bound is used to ensure well-foundedness in the second case, for X_i classified by a type. The issue is that we need to stipulate that if X normalizes to $(c_i\ \bar{X})$, the X_i are all reducible, since some of these may be proofs of non-atomic formulas. This situation does not arise for Gödel's System T, where data cannot contain functions. But then, we cannot just insist that the X_i are reducible at their types, since a piece of data of type `list`, for example, can easily contain subdata of the same type. The definition of reducibility would fail to be well-founded. The data bound gets around this difficulty.

We call an expression reducible with classifier A (without the data bound) if there exists a data bound such that the expression is reducible of classifier A with that data bound. Recall that our well-founded classifier ordering makes instances of a quantified formula (like the instance F' in the clause above for existentials) smaller than the quantified formula. We have also made datatypes smaller than all non-type classifiers that their constructors can depend on. So this definition of *RED* is well-founded. Note that the definition differs from the PaT in its treatment of datatypes. This is to allow term constructors to take proofs as arguments. The corresponding property is not allowed by Gödel's System T.

We now define the neutral expressions to be variables and logical eliminations, namely proof-level applications, `induction`-redexes, and existential eliminations. Closely following PaT, we must now prove the three critical properties:

1. If $X \in \mathit{RED}_A$, then X is strongly normalizing.

2. If $X \in \mathit{RED}_A$ and $X \rightsquigarrow X'$, then $X' \in \mathit{RED}_A$.

3. If $X^*[R]$ is neutral, $R \rightsquigarrow C$, and $X^*[C] \in \mathit{RED}_A$; then $X \in \mathit{RED}_A$.

The three properties are established by the same (easy) reasoning for atomic classifiers, existentials, and universals as in PaT, with existentials handled by the same reasoning as for product types. For types, the properties follow easily, as in the case for atomic classifiers. Again using the same reasoning as in PaT, we may then show that `existsi` and `foralli` preserve reducibility. Application of term and type constructors are also easily seen to preserve reducibility. Cast terms also preserve reducibility. The critical point there is that we cannot cast from a type to a non-type classifier, or vice versa, and so if t is reducible of type T, then it is also reducible of type T'. If there were a possibility to cast from a type to, say, a `Forall`-formula, the proof would not go through at this point.

Finally, we must show that `induction`-redexes preserve reducibility. We must observe that cast shifting preserves reducibility (in particular, `Cast` does). So we assume that the body P_i of each case in the `induction` proof is reducible, for all reducible values that can be substituted for the pattern variables. As in the PaT proof (Section 7.2 of [12]), we prove that all terms reachable in one reduction step are reducible. The third property then allows us to conclude the `induction`-redex is reducible. We reason by induction on a different norm than the one used in the PaT proof, to accomodate our richer datatypes. The norm is the lexicographic combination of (a) the maximum number of constructors in the normal forms (we do not assume confluence) of the scrutinized term and (b) the sum of the maximum lengths of the normalization sequences for the arguments to the `induction`-proof and those lengths for the bodies of the cases. The essential case is when the `induction`-redex itself reduces, in which case new `induction`-redexes may be created. But their norms are smaller, since the first component has decreased. By the definition of reducibility for types, all the subdata of the (normal) scrutinized term are reducible, so reducibility of the body of the taken case follows.

From these considerations, it is now straightforward to conclude, by a similar inductive argument as in PaT:

Theorem 6.2. (Strong Normalization)
All well-typed expressions are reducible, and hence by the first property, strongly normalizing.

7. Equational Soundness and Consistency

Having eliminated logical cuts, we proceed now to show consistency of the system. This will be done with the help of a characterization of equalities and disequalities provable in equational contexts: that is, the only free variables allowed (in the typing context) are ones proving equations. We simultaneously prove the following three assertions.

Theorem 7.1. (Consistency)
For all proofs P classifiable in the empty context and in normal form with respect to the proof reduction of the previous section, the following hold.

1. P does not prove `False`.

2. If P proves an equation $\{t = t'\}$ or $\{T = T'\}$, then one of the following is true:

(a) t and t' are both diverging.

(b) t and t' are joinable at a value or stuck term.

(c) 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.

3. If P proves a disequation $\{t \texttt{ != } t'\}$ or $\{T \texttt{ != } T'\}$, then one of the following holds:

(a) t and t' evaluate to values of the form $C[I_1, \bar{I}]$ and $C'[I_2, \bar{I'}]$, where C and C' have different heads.

(b) T and T' are of the forms $C[I_1, \bar{I}]$ and $C'[I_2, \bar{I'}]$, where C and C' have different heads and again some of the $\bar{I}, \bar{I'}$ may be stuck.

(c) T is a functional type and T' is of the form $\langle d\ \bar{Y} \rangle$, or vice versa.

The proof is by induction on the structure of P in normal form. We must first observe that P cannot be a stuck `induction`-redex. Such could in principle arise if the scrutinized argument contained an intervening cast that could not be shifted. A cast can fail to be shifted if the term t being cast to functional type does not itself have functional type. But in that case, the cast's proof P' would have to prove that some non-functional type is equal to a functional type. In the empty context, that cannot happen unless the non-functional type is a type application $\langle d\ \bar{Y} \rangle$. By induction, P' cannot prove an equation of this form, as it violates the characterization of equations provable in the empty context. Also, an `existse1`-term cannot cause an `induction`-redex to be stuck, since in the empty context, such could only be applied to an `existsi`-proof, violating the assumption that P is normal. So normal P cannot be a stuck `induction`-redex. Similar standard reasoning shows that P cannot prove `False` if it ends in a logical elimination rule, and it obviously cannot prove `False` if it ends in a logical elimination rule. It could *a priori* prove `False` if it ends in a `contra` inference, but the disjointness of our characterizations of the pairs of provably equal terms (or types) and the pairs of provably disequal ones prevents, by induction, such an inference. This shows the first part of the Theorem.

For the second and third parts, we simply observe that the characterization is preserved by our equational inferences (Figure 9), and that by induction, P cannot end in `falsee`. The `trans` case, of course, has the most subcases to check. For example, suppose we are going from $\{t_1 \texttt{ = } t_2\}$ and $\{t_2 \texttt{ != } t_3\}$ to $\{t_1 \texttt{ != } t_3\}$. An example subcase is then if t_2 and t_3 evalute to differently headed inactive terms I_1 and I_2. It cannot happen that t_1 and t_2 are diverging, so it must be that they are joinable at a value. By determinism of the operational semantics, this value must be the same as the I_1, and hence differently headed from I_2. Thus the characterization is maintained for the derived disequation $\{t_1 \texttt{ != } t_3\}$. This concludes the proof.

8. Type Soundness for Typed Term Reduction

We now prove type soundness for a typed version of the operational semantics (Figure 4) of OpTT. Rules for reducing redexes in this typed operational semantics are given in Figure 12, and rules for reduction in context are given in Figure 13. Only well-typed terms are to be evaluated, and only in the empty context. All rules are to be applied modulo cast shifting, defined just as above for proof reduction. By Theorem 7.1, cast shifting cannot get stuck, because all types provably equal in the empty context have

$$
\begin{array}{lll}
(F\ \bar{V}) & \leadsto & [\bar{V}/\bar{x}, F/x]t \\
F\ \equiv\ \texttt{fun}\ x(\bar{x} : \bar{A})\ :\ T\ .t & & \\
\\
\texttt{match cast}\ (c_i\ \bar{V})\ \texttt{by}\ P\ \texttt{by}\ x\ y\ \texttt{with}\ C_1 | \ldots | C_n \texttt{end} & \leadsto & [\bar{V}/\bar{x}_i, \texttt{refl}\ (c_i\ \bar{V})/x, \texttt{symm}\ P/y]s_i \\
\forall i.(C_i \equiv c_i\ \bar{x}_i \texttt{=>} s_i) & & \\
\\
\texttt{let}\ x\ =\ V\ \texttt{by}\ y\ \texttt{in}\ t & \leadsto & [V/x, \texttt{refl}\ V/y]t
\end{array}
$$

where:

$$V \quad ::= \quad x \mid\mid c \mid\mid T \mid\mid P \mid\mid (c\ V_1\ \cdots\ V_n) \mid\mid \texttt{fun}\ x(\bar{x}\ :\ \bar{A})\ :T.\ t \mid\mid \texttt{cast}\ V\ \texttt{by}\ P$$

Figure 12. Typed Redex Reductions for Terms

the same top-level form. We cannot equate a type-level application and a Fun-type, by Theorem 7.1. Also, in the empty context, we also cannot cast a term using an equation between a type variable and another expression. The only values of functional type are (casts of) fun-terms, and the only values of type $\langle d\ \bar{Y} \rangle$ are (casts of) applications of constructors for d. So term-level eliminations cannot get stuck (modulo cast shifting) in the empty context.

We may verify by induction that if $t \leadsto t'$ in this typed operational semantics, then $|t| \leadsto |t'|$ in the untyped operational semantics (Figure 4). This justifies the use of evalstep in the first rule of Figure 13. That rule is present, of course, to account for dependence of the type of a term on a redex in it. We may then easily verify type preservation by induction on the structure of the $\leadsto$-derivation. These considerations justify:

Theorem 8.1. (Type Preservation)
If $t \leadsto t'$ with $t : T$, then $t' : T$.

Theorem 8.2. (Progress)
If $t : T$, then either t is a value or there exists t' with $t \leadsto t'$.

9. 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. This paper has studied the meta-theory of a core OPTT. As designed, the

$$\frac{t_2 \rightsquigarrow t_2' \qquad V : \mathtt{Fun}(x : A).T^*[x]}{(V\ t_2) \rightsquigarrow \mathtt{cast}\ (V\ t_2')\ \mathtt{by}\ \mathtt{cong}\ T^*\ \mathtt{evalstep}\ t_2\ t_2'}$$

$$\frac{t \rightsquigarrow t'}{E[t] \rightsquigarrow E[t']}$$

$$\begin{array}{rcl} E & ::= & * \mid\mid (E\ X) \mid\mid \mathtt{let}\ x\ =\ E\ \mathtt{by}\ y\ \mathtt{in}\ t \mid\mid \\ & & \mathtt{match}\ E\ \mathtt{by}\ x\ y\ \mathtt{with}\ c_1\ \bar{x}_1\ \mathtt{=>}\ t_1 \mid \ldots \mid c_n\ \bar{x}_n\ \mathtt{=>}\ t_n\ \mathtt{end} \end{array}$$

Figure 13. Typed Reductions for Terms

meta-theoretic development is simple, in the proof theoretic sense. Future work includes further extensions to the language to accomodate more features of practical programming languages such as mutable state, input/output, and global variables. Non-functional features can be accomodated using functional modeling. A functional model for the non-functional feature is devised and used for formal reasoning about programs. This model is replaced during compilation with the efficient non-functional implementation. Correctness of this replacement is outside the verification environment, and must be trusted. Soundness of the approach can be enforced using linear types to control resource usage. A similar idea is proposed in [24].

Acknowledgements: Thorsten Altenkirch for detailed comments on an earlier draft, and the NSF for support under award CCF-0448275.

References

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

[2] Altenkirch, T., McBride, C., Swierstra, W.: Observational Equality, Now!, *PLPV '07: Proceedings of the 2007 Workshop on Programming Languages meets Program Verification* (A. Stump, H. Xi, Eds.), 2007.

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

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

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

[6] Bove, A., Capretta, V.: Modelling General Recursion in Type Theory, *Mathematical Structures in Computer Science*, **15**, February 2005, 671–708, Cambridge University Press.

[7] de Bruijn, N.: *A plea for weaker frameworks*, in: Huet and Plotkin [14], 1991, 40–67.

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

[9] Chapman, J.: Type Theory Should Eat Itself, *Proc. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice* (A. Abel, C. Urban, Eds.), 2008.

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

[11] Constable, R., Smith, S.: Partial Objects in Constructive Type Theory, *Proceedings of the Symposium on Logic in Computer Science*, 1987.

[12] Girard, J.-Y., Lafont, Y., Taylor, P.: *Proofs and Types*, Cambridge University Press, 1989.

[13] Hofmann, M., Streicher, T.: The groupoid interpretation of type theory, *Twenty-five years of constructive type theory* (G. Sambin, Ed.), Oxford: Clarendon Press, 1998.

[14] Huet, G., Plotkin, G., Eds.: *Logical Frameworks*, Cambridge University Press, 1991.

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

[16] McBride, C.: *Dependently Typed Functional Programs and Their Proofs*, Ph.D. Thesis, 1999.

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

[18] Nanevski, A., Morrisett, G.: *Dependent Type Theory of Stateful Higher-Order Functions*, Technical Report TR-24-05, Harvard University, 2005.

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

[20] Rabe, F.: First-Order Logic with Dependent Types, *Proceedings of the 3rd International Joint Conference on Automated Reasoning* (N. Shankar, U. Furbach, Eds.), 4130, Springer, 2006.

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

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

[23] Stirling, C.: A game-theoretic approach to deciding higher-order matching, *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2006.

[24] Swierstra, W., Altenkirch, T.: Beauty in the Beast, *Haskell Workshop*, 2007.