Combining Programming with Theorem Proving *

Chiyan Chen Hongwei Xi

Boston University

{chiyan, hwxi}@cs.bu.edu

Abstract

Applied Type System ($\mathcal{ATS}$) is recently proposed as a framework for designing and formalizing (advanced) type systems in support of practical programming. In $\mathcal{ATS}$, the definition of type equality involves a constraint relation, which may or may not be algorithmically decidable. To support practical programming, we adopted a design in the past that imposes certain restrictions on the syntactic form of constraints so that some effective means can be found for solving constraints automatically. Evidently, this is a rather *ad hoc* design in its nature. In this paper, we rectify the situation by presenting a fundamentally different design, which we claim to be both novel and practical. Instead of imposing syntactical restrictions on constraints, we provide a means for the programmer to construct proofs that attest to the validity of constraints. In particular, we are to accommodate a programming paradigm that enables the programmer to combine programming with theorem proving. Also we present some concrete examples in support of the practicality of this design.

Categories and Subject Descriptors D.3 [*Software*]: Programming Languages

General Terms Languages, Verification

Keywords ATS, Applied Type System, Dependent Types, Proof Erasure, Theorem Proving

1. Introduction

The notion of type equality plays a pivotal rôle in type system design. However, the importance of this role is often less evident in commonly studied type systems. For instance, in the simply typed λ-calculus, two types are considered equal if and only if they are syntactically the same; in the second-order polymorphic λ-calculus, two types are considered equal if and only if they are α-equivalent; in the higher-order polymorphic λ-calculus, two types are considered equal if and only if they are $\beta\eta$-equivalent. The situation immediately changes in the framework *Applied Type System* ($\mathcal{ATS}$) [25, 27], and we now use a simple example to stress this point.

In Figure 1, we implement a function in ATS (via a form of syntax rather similar to that of Standard ML [12]), where ATS is

* Partially supported by NSF grant no. CCR-0229480

ICFP'05 September 26–28, 2005, Tallinn, Estonia.


```
fun append {a:type, m:nat, n:nat}
   (xs: list(a,m), ys: list(a,n)): list(a,m+n) =
  case xs of
    | nil => ys (* the first clause *)
    | cons (x, xs) => (* the second clause *)
      cons (x, append (xs, ys))
```

Figure 1. A simple function in ATS

a programming language with its type system rooted in $\mathcal{ATS}$. We use $\mathbf{list}$ as a type constructor; when applied to a type T and an integer I, $\mathbf{list}(T, I)$ forms a type for lists of length I in which each element is of type T. Also, the two list constructors *nil* and *cons* are assigned the following types:

$$\begin{array}{rcl} \mathit{nil} & : & \forall a : \mathit{type}.\mathbf{list}(a, 0) \\ \mathit{cons} & : & \forall a : \mathit{type}.\forall n : \mathit{nat}.(a, \mathbf{list}(a, n)) \rightarrow \mathbf{list}(a, n+1) \end{array}$$

The header of the function append indicates that append is assigned the following type:

$$\begin{array}{l} \forall a : \mathit{type}.\forall m : \mathit{nat}.\forall n : \mathit{nat}. \\ \quad (\mathbf{list}(a, m), \mathbf{list}(a, n)) \rightarrow \mathbf{list}(a, m+n) \end{array}$$

which means that append returns a list of length $m + n$ when applied to two lists of length m and n, respectively. Note that *type* is a built-in sort in $\mathcal{ATS}$, and a static term of the sort *type* stands for a type. Also, int is a built-in sort for integers in $\mathcal{ATS}$, and nat is an abbreviation of the subset sort $\{a : \mathit{int} \mid a \geq 0\}$ for all nonnegative integers.

When type-checking the definition of append, we essentially need to generate the following two constraints:

$$\begin{array}{ll} 1. & \forall m : \mathit{nat}.\forall n : \mathit{nat}.m = 0 \supset n = m + n \\ 2. & \forall m : \mathit{nat}.\forall n : \mathit{nat}.\forall m' : \mathit{nat}. \\ & \quad m = m' + 1 \supset (m' + n) + 1 = m + n \end{array}$$

The first constraint is generated when the first clause is type-checked, which is needed for determining that the types $\mathbf{list}(a, n)$ and $\mathbf{list}(a, m+n)$ are equal under the assumption that $\mathbf{list}(a, m)$ equals $\mathbf{list}(a, 0)$. Similarly, the second constraint is generated when the second clause is type-checked, which is needed for determining that the types $\mathbf{list}(a, (m'+n)+1)$ and $\mathbf{list}(a, m+n)$ are equal under the assumption that $\mathbf{list}(a, m)$ equals $\mathbf{list}(a, m'+1)$. Clearly, we need to impose certain restrictions on the form of constraints allowed in practice so that an effective means can be found to solve constraints. In ATS, we require that (arithmetic) constraints like those presented above be linear,[1] and we rely on a constraint solver based on the Fourier-Motzkin variable elimination method [6] to solve such constraints. While this is indeed a

[1] More precisely, we require that an arithmetic constraint can be turned into a linear integer programming problem.

```
datasort mynat = Z | S of mynat

datatype myadd (mynat, mynat, mynat) =
  | {n: mynat} Bas (Z, n, n)
  | {m: mynat, n: mynat, s: mynat}
      Ind (S m, n, S s) of myadd (m, n, s)

datatype mylist (type, mynat) =
  | {a: type} mynil (a, Z)
  | {a: type, n: mynat}
      mycons (a, S n) of (a, mylist (a, n))

// {...} : universal quantifier
// [...] : existential quantifier
fun myappend {a:type, m:mynat, n:mynat, s:mynat}
   (xs: mylist (a, m), ys: mylist (a, n))
  : [s: mynat] '(myadd (m, n, s), mylist (a, s)) =
  case xs of
    | mynil => '(Bas, ys)
    | mycons (x, xs) =>
      let val '(pf, zs) = myappend (xs, ys) in
         '(Ind pf, mycons (x, zs))
      end
```

Figure 2. A motivating example

simple design, it can also be too restrictive, sometimes, in a situation where nonlinear constraints (e.g., $\forall n : int.\ n * n \geq 0$) need to be handled. Furthermore, such a design is inherently *ad hoc* in its nature.

In this paper, we present a fundamentally different design. We are to provide a means for the programmer to handle nonlinear constraints by constructing explicit proofs (while linear constraints are still solved by a constraint solver). For a simpler presentation, let us assume for this moment that even the addition function on integers is not allowed in forming constraints. Under such a restriction, we can still implement a list append function that is assigned a type capturing the invariant that the length of the concatenation of two given lists xs and ys equals $m + n$ if xs and ys are of length m and n, respectively. For instance, this is achieved by the code in Figure 2, and we provide some explanation for it as follows.

In Figure 2, we first declare a datasort $mynat$ for forming terms that can be used as type index expressions. We then declare a datatype constructor $\mathbf{myadd}$ such that $\mathbf{myadd}$ forms a type $\mathbf{myadd}(s_1, s_2, s_3)$ when applied to three static terms s_1, s_2, s_3 of the sort $mynat$. The syntax indicates that there are two value constructors associated with $\mathbf{myadd}$, which are given the following types:

$$\begin{array}{lll} \textit{Bas} & : & \forall a : mynat.\ \mathbf{myadd}(Z, a, a) \\ \textit{Ind} & : & \forall a_1 : mynat.\forall a_2 : mynat.\forall a_3 : mynat. \\ & & \quad \mathbf{myadd}(a_1, a_2, a_3) \rightarrow \mathbf{myadd}(S(a_1), a_2, S(a_3)) \end{array}$$

Given static terms s_1, s_2, s_3 of the sort $mynat$, it is easy to see that there exists a closed value of the type $\mathbf{myadd}(s_1, s_2, s_3)$ if and only if $|s_1| + |s_2| = |s_3|$, where we use $|s|$ for the number of occurrences of S in s (which is assumed to be closed). We next declare a datatype constructor $\mathbf{mylist}$, which forms a type $\mathbf{mylist}(T, s)$ when applied to a type T and a term s of the sort $mynat$. Note that the two constructors *mynil* and *mycons* are assigned the following types:

$$\begin{array}{lll} \textit{mynil} & : & \forall a : type.\ \mathbf{mylist}(a, Z) \\ \textit{mycons} & : & \forall a : type.\forall n : mynat. \\ & & \quad (a, \mathbf{mylist}(a, n)) \rightarrow \mathbf{mylist}(a, S(n)) \end{array}$$

Thus, given a type T and a term s of the sort $mynat$, the type $\mathbf{mylist}(T, s)$ is for lists of length $|s|$ in which each element is of type T. Lastly, we define a function *myappend*, which is given the following type:

$$\begin{array}{l} \forall a : type.\forall a_1 : mynat.\forall a_2 : mynat. \\ \quad (\mathbf{mylist}(a, a_1), \mathbf{mylist}(a, a_2)) \rightarrow \\ \quad \exists a_3 : mynat.(\mathbf{myadd}(a_1, a_2, a_3), \mathbf{mylist}(a, a_3)) \end{array}$$

Note that we use $'(\ldots)$ in the concrete syntax to form tuple types as well as tuples, and the quote symbol ($'$) is solely for the purpose of parsing. Given two lists xs and ys of types $\mathbf{mylist}(T, s_1)$ and $\mathbf{mylist}(T, s_2)$, respectively, for some type T and terms s_1 and s_2 of the sort $mynat$, *myappend* returns a pair (pf, zs) such that *pf* is a value of type $\mathbf{myadd}(s_1, s_2, s_3)$ for some term s_3 of the sort $mynat$ and *zs* is a list of type $\mathbf{mylist}(a, s_3)$; the value *pf*, which we call a *witnessing value*, essentially serves as a witness to the fact that $|s_3| = |s_1| + |s_2|$, that is, the length of zs is the sum of the lengths of xs and ys.

So far, what we have described can already be implemented in Dependent ML (DML) [21]. Certainly, the programming style as is presented in Figure 2 is more involved than the usual functional programming style. However, this is not so much a concern as we expect to make only occasional use of this programming style. In particular, we emphasize that the programmer can choose not to program in such a style by simply avoiding capturing certain program invariants. What is of real concern is the need for constructing witnessing values (e.g., *pf* in the definition of *myappend*) at run-time. For instance, we have a realistic example (array subscripting function) where the underlying algorithm is $O(1)$-time but an involved witnessing value takes $O(n)$-time to construct. This is simply unacceptable in practice.

The primary contribution of the paper lies in the novel programming language design we propose that allows programs to be combined with proofs while obviating the need for constructing witnessing values at run-time. With this design, we save not only time but also space when evaluating programs (that contain proofs). More importantly, we become able to verify at compile-time the correctness of witnessing values, that is, these values indeed witness the facts they are supposed to witness. Though we only combine programming with proofs from a particular proof system (based on intuitionistic predicate logic) in this paper, we stress that the design itself is general and flexible in its nature. For instance, we also support in ATS the construction of proofs based a form of linear logic (closely related to separation logic [17] for establishing properties on memory) [30]. In support of the practicality of this design, we have finished a running implementation of ATS [27] and written tens of thousands lines of code in ATS itself[2], where a significant part is involved, either directly or indirectly, with proof construction.

At this point, we stress that this design for combining programming with theorem proving is fundamentally different from the programming paradigm (as is supported in certain theorem proving systems such as NuPrl [4] and Coq [7]) in which (total) programs are extracted out of proofs. In ATS, program construction may involve programming constructs such as general recursion and nonexhaustive pattern matching that are in principle not allowed in proof construction. The distinction between proofs and programs we propose is partly inspired by the distinction between logical parts and informative parts employed in extracting programs out of proofs in Coq [14]. However, there is also a profound difference: We allow proofs in programs but not programs in proofs while logical parts may contain informative parts and vice versa. In partic-

[2] The library of ATS alone already contains more than 20,000 lines of code in ATS at this moment.

ular, we extract nothing out of proofs, which are simply erased at run-time. This will be further illustrated later with some concrete examples.

Also, we emphasize that combining programming with theorem proving is not just a simple matter of hooking up programming languages with (automated) theorem provers. After all, we have so far not seen it done elsewhere effectively in practice. Thus, we consider a design that actually supports practical programming with theorem programming to be an important contribution.

The rest of the paper is organized as follows. In Section 2, we demonstrate an approach to combining programs with proofs in the design and formalization of a language λ_{pf}, setting up some machinery for further development. In order to reap the benefits of combining programs with proofs, we extend λ_{pf} to $\lambda_{pf}^{\forall,\exists}$ in Section 3 by introducing universally as well as existentially quantified types. We then present a few examples in Section 4 to give the reader some concrete feel as to how the approach to combining programs with proofs can be applied in practice. Lastly, we mention some related work and conclude.

There is a full version of the paper available on-line [1] in which more details such as proofs and examples can be found.

$$\frac{}{\Pi, \underline{x} : P \vdash \underline{x} : P}\ \textbf{(pr-var)}$$

$$\frac{}{\Pi \vdash \langle\rangle : \mathbf{1}}\ \textbf{(pr-unit)}$$

$$\frac{\Pi \vdash \underline{d}_1 : P_1 \quad \Pi \vdash \underline{d}_2 : P_2}{\Pi \vdash \langle \underline{d}_1, \underline{d}_2 \rangle : P_1 * P_2}\ \textbf{(pr-tup)}$$

$$\frac{\Pi \vdash \underline{d} : P_1 * P_2}{\Pi \vdash \mathbf{fst}(\underline{d}) : P_1}\ \textbf{(pr-fst)}$$

$$\frac{\Pi \vdash \underline{d} : P_1 * P_2}{\Pi \vdash \mathbf{snd}(\underline{d}) : P_2}\ \textbf{(pr-snd)}$$

$$\frac{\Pi, \underline{x} : P_1 \vdash \underline{d} : P_2}{\Pi \vdash \mathbf{lam}\ \underline{x}.\underline{d} : P_1 \to P_2}\ \textbf{(pr-lam)}$$

$$\frac{\Pi \vdash \underline{d}_1 : P_1 \to P_2 \quad \Pi \vdash \underline{d}_2 : P_1}{\Pi \vdash \mathbf{app}(\underline{d}_1, \underline{d}_2) : P_2}\ \textbf{(pr-app)}$$

Figure 4. The rules for assigning props to proof terms in λ_{pf}

2. Formal Development

In this section, we present a typed language λ_{pf}, formally demonstrating a design for combining programs with proofs. The language λ_{pf}, which is essentially built on top of the simply typed λ-calculus, is not intended for demonstrating some practical applications of combining programs with proofs as such applications are difficult to find until dependent types are introduced. The primary purpose of λ_{pf} is to set up the machinery needed for further development.

The syntax of λ_{pf} is given in Figure 3. There are proof terms and dynamic terms in λ_{pf}, and we are to present rules for assigning types to these terms. In order to avoid potential confusion, the types for proof terms are called *props*. We use P for props, $\underline{d}$ for proof terms and $\underline{v}$ for proof values. Also, we use Π for contexts in which proof variables are declared. The rules for assigning props to proof terms are given in Figure 4, where we use a judgment of the form $\Pi \vdash \underline{d} : P$ to mean that $\underline{d}$ can be given the prop P under the context Π. We use T for types, d for dynamic terms and v for dynamic values. There are two forms of dynamic variables: x and f; we use the names *lam-variable* and *fix-variable* to refer to x and f, respectively; the former is a value while the latter is not. We may write xf to mean either a lam-variable or a fix-variable.

Intuitively, a type of the form $P * T$ is to be assigned to a value of the form $\langle \underline{v}, v\rangle$ such that $\underline{v}$ is a proof value of prop P and v is a dynamic value of type T; therefore, if a value of type $P * T$ is produced, then we know that the prop P holds. Also, a type of the form $P \to T$ is to be assigned to a value of the form **lam** $\underline{x}.v$, which can only be of use if a proof of prop P is made available. For those who are familiar with the recently proposed framework $\mathcal{ATS}$ [25, 27], these two forms of types are closely related to but different from asserting types and guarded types in $\mathcal{ATS}$.

The typing judgment in λ_{pf} is of the form $\Pi; \Delta \vdash d : T$, where we use Δ for contexts in which dynamic variables are declared, and the rules for deriving such typing judgments are given in Figure 5.

We now assign dynamic semantics to dynamic terms. For doing so, we also need to assign dynamic semantics to proof terms. As usual, we first introduce the notion of evaluation contexts in Figure 3. Given a proof evaluation context $\underline{E}$, we write $\underline{E}[\underline{d}]$ for the proof term obtained from replacing the hole $\underline{[]}$ in $\underline{E}$ with $\underline{d}$. Given a dynamic evaluation context E, we know that it contains a hole which is either $\underline{[]}$ or $[]$; in the former case, we write $E[\underline{d}]$ for the dynamic term obtained from replacing $\underline{[]}$ in E with $\underline{d}$; in the latter case, we we write $E[d]$ for the dynamic term obtained from replacing $[]$ in E with d. We next introduce proof redexes and dynamic redexes.

DEFINITION 2.1. *We define proof redexes and dynamic redexes as follows.*

- $\mathbf{fst}(\langle \underline{v}_1, \underline{v}_2\rangle)$ *is a proof redex, and its reduction is* $\underline{v}_1$.
- $\mathbf{snd}(\langle \underline{v}_1, \underline{v}_2\rangle)$ *is a proof redex, and its reduction is* $\underline{v}_2$.
- $\mathbf{app}(\mathbf{lam}\ \underline{x}.\underline{d}, \underline{v})$ *is a proof redex, and its reduction is* $\underline{d}[\underline{x} \mapsto \underline{v}]$.
- **let** $\langle \underline{x}, x\rangle = \langle \underline{v}, v\rangle$ **in** d *is a dynamic redex, and its reduction is* $d[\underline{x} \mapsto \underline{v}][x \mapsto v]$.
- $\mathbf{app}(\mathbf{lam}\ \underline{x}.d, \underline{v})$ *is a dynamic redex, and its reduction is* $d[\underline{x} \mapsto \underline{v}]$.
- **let** $\underline{x} = \underline{v}$ **in** d *is a dynamic redex, and its reduction is* $d[\underline{x} \mapsto \underline{v}]$.
- $\mathbf{fst}(\langle v_1, v_2\rangle)$ *is a dynamic redex, and its reduction is* v_1.
- $\mathbf{snd}(\langle v_1, v_2\rangle)$ *is a dynamic redex, and its reduction is* v_2
- $\mathbf{app}(\mathbf{lam}\ x.d, v)$ *is a dynamic redex, and its reduction is* $d[x \mapsto v]$.
- **let** $x = v$ **in** d *is a dynamic redex, and its reduction is* $d[x \mapsto v]$.
- **fix** $f.d$ *is a dynamic redex, and its reduction is* $d[f \mapsto$ **fix** $f.d]$.

We leave out the details on the (standard) substitution involved in the above definition.

Given $\underline{d}_1$ and $\underline{d}_2$ such that $\underline{d}_1 = \underline{E}[\underline{d}]$ and $\underline{d}_2 = \underline{E}[\underline{d}']$ for some proof redex $\underline{d}$ and its reduction $\underline{d}'$, we write $\underline{d}_1 \underline{\to} \underline{d}_2$ and say that $\underline{d}_1$ reduces to $\underline{d}_2$ in one step. Given d_1 and d_2, we write $d_1 \to d_2$ and say that d_1 reduces to d_2 in one step if (1) $d_1 = E[\underline{d}_1]$ and $d_2 = E[\underline{d}_2]$ for some $\underline{d}_1 \underline{\to} \underline{d}_2$ or (2) $d_1 = E[d]$ and $d_2 = E[d']$ for some dynamic redex d and its reduction d'. We use $\underline{\to}^*$ and $\to^*$ for the reflexive and transitive closures of $\underline{\to}$ and $\to$, respectively.

The type soundness of λ_{pf} can be established in a standard manner, and some of the lemmas and theorems involved are given as follows. Please see [1] for details on proofs.

LEMMA 2.2 (Substitution). *We have the following:*

1. *Assume that* $\Pi \vdash \underline{d}_1 : P_1$ *and* $\Pi, \underline{x} : P_1 \vdash \underline{d}_2 : P_2$ *are derivable. Then* $\Pi \vdash \underline{d}_2[\underline{x} \mapsto \underline{d}_1] : P_2$ *is also derivable.*

$$
\begin{array}{llll}
\text{props} & P & ::= & \mathbf{1} \mid P_1 * P_2 \mid P_1 \rightarrow P_2 \\
\text{proof terms} & \underline{d} & ::= & \underline{x} \mid \langle\rangle \mid \langle \underline{d}_1, \underline{d}_2 \rangle \mid \textbf{fst}(\underline{d}) \mid \textbf{snd}(\underline{d}) \mid \textbf{lam}\ \underline{x}.\underline{d} \mid \textbf{app}(\underline{d}_1, \underline{d}_2) \\
\text{proof values} & \underline{v} & ::= & \underline{x} \mid \langle \underline{v}_1, \underline{v}_2 \rangle \mid \textbf{lam}\ \underline{x}.\underline{d} \\
\text{proof var. ctx.} & \Pi & ::= & \emptyset \mid \Pi, \underline{x} : P \\
\text{types} & T & ::= & \mathbf{1} \mid P * T \mid P \rightarrow T \mid T_1 * T_2 \mid T_1 \rightarrow T_2 \\
\text{dynamic terms} & d & ::= & x \mid f \mid \langle\rangle \mid \langle \underline{d}, d \rangle \mid \textbf{let}\ \langle \underline{x}, x \rangle = d_1\ \textbf{in}\ d_2 \mid \\
& & & \textbf{lam}\ \underline{x}.v \mid \textbf{app}(d, \underline{d}) \mid \textbf{let}\ \underline{x} = \underline{d}\ \textbf{in}\ d \mid \\
& & & \langle d_1, d_2 \rangle \mid \textbf{fst}(d) \mid \textbf{snd}(d) \mid \textbf{lam}\ x.d \mid \textbf{app}(d_1, d_2) \mid \\
& & & \textbf{let}\ x = d_1\ \textbf{in}\ d_2 \mid \textbf{fix}\ f.d \\
\text{dynamic values} & v & ::= & x \mid \langle \underline{v}, v \rangle \mid \textbf{lam}\ \underline{x}.v \mid \langle v_1, v_2 \rangle \mid \textbf{lam}\ x.d \\
\text{dynamic var. ctx.} & \Delta & ::= & \emptyset \mid \Delta, x : T \\
\text{proof eval. ctx.} & \underline{E} & ::= & \underline{[]} \mid \langle \underline{E}, \underline{d} \rangle \mid \langle \underline{v}, \underline{E} \rangle \mid \textbf{fst}(\underline{E}) \mid \textbf{snd}(\underline{E}) \mid \textbf{app}(\underline{E}, \underline{d}) \mid \textbf{app}(\underline{v}, \underline{E}) \\
\text{dynamic eval. ctx.} & E & ::= & \underline{[]} \mid \langle \underline{[]}, d \rangle \mid \langle \underline{v}, E \rangle \mid \textbf{let}\ \langle \underline{x}, x \rangle = E\ \textbf{in}\ d \mid \textbf{app}(E, \underline{d}) \mid \textbf{app}(v, \underline{[]}) \mid \textbf{let}\ \underline{x} = \underline{[]}\ \textbf{in}\ d \\
& & & \langle E, d \rangle \mid \langle v, E \rangle \mid \textbf{fst}(E) \mid \textbf{snd}(E) \mid \textbf{app}(E, d) \mid \textbf{app}(v, E) \mid \textbf{let}\ x = E\ \textbf{in}\ d
\end{array}
$$

Figure 3. The syntax for λ_{pf}

2. *Assume that* $\Pi \vdash \underline{d} : P$ *and* $\Pi, \underline{x} : P; \Delta \vdash d : T$ *are derivable. Then* $\Pi; \Delta \vdash d[\underline{x} \mapsto \underline{d}] : T$ *is also derivable.*
3. *Assume that* $\Pi; \Delta \vdash d_1 : T_1$ *and* $\Pi; \Delta, x : T_1 \vdash d_2 : T_2$ *are derivable. Then* $\Pi; \Delta \vdash d_2[x \mapsto d_1] : T_2$ *is also derivable.*

THEOREM 2.3 (Totality). *Assume that* $\emptyset \vdash \underline{d} : P$ *is derivable. Then* $\underline{d} \underline{\rightarrow}^* \underline{v}$ *holds for some proof value* $\underline{v}$ *of prop* P.

THEOREM 2.4 (Subject Reduction). *Assume that* $\emptyset; \emptyset \vdash d : T$ *is derivable and* $d \rightarrow d'$ *holds. Then* $\emptyset; \emptyset \vdash d' : T$ *is also derivable.*

THEOREM 2.5 (Progress). *Assume that* $\emptyset; \emptyset \vdash d : T$ *is derivable. Then either* d *is a value or* $d \rightarrow d'$ *holds for some dynamic term* d'.

We are now ready to establish a key property of λ_{pf}, which states that proof terms *cannot* affect the dynamic semantics of a dynamic term. We first introduce an erasure function in Figure 6, which erases all syntax related to proof terms in a given dynamic term. The following theorem indicates that the evaluation of a well-typed closed dynamic term d can be performed by simply evaluating the erasure of d, thus obviating the need for constructing proof values at run-time.

THEOREM 2.6. *Assume that* $\emptyset; \emptyset \vdash d : T$ *is derivable.*

1. *If* $d \rightarrow^* v$, *then* $|d| \rightarrow^* |v|$.
2. *If* $|d| \rightarrow^* v$, *then* $d \rightarrow^* v'$ *for some dynamic value* v' *such that* $|v'| = v$.

Note that Theorem 2.3 plays a crucial rôle in the proof of Theorem 2.6.

3. Extension

While the basic design for combining programs with proofs is already demonstrated in the formalization of λ_{pf}, it is nonetheless difficult to truly reap the benefits of this design given that the type system of λ_{pf} is simply too limited. We now extend λ_{pf} to $\lambda_{pf}^{\forall,\exists}$ with universally as well as existentially quantified types. Following the work in [25, 27], we present in the rest of this section an overview of this extension.

Like an applied type system [25], that is, a type system formed in the framework $\mathcal{ATS}$, $\lambda_{pf}^{\forall,\exists}$ consists of a static component (statics) and a dynamic component (dynamics). The (additional) syntax for $\lambda_{pf}^{\forall,\exists}$ is given in Figure 7. The statics itself is a simply typed language and a type in it is called *sort*. We assume the existence of the following basic sorts: $bool$, int, $prop$ and $type$; $bool$ is the sort for truth values, and int is the sort for integers, and $prop$ is the sort for props, and $type$ is the sort for types. We use a for static variables, b for truth values *tt* and *ff*, and i for integers. A term s in the statics is called a static term, and we write $\Sigma \vdash s : \sigma$ to mean that s can be given the sort σ under the context Σ, which assigns sorts to static variables. The rules for assigning sorts to static terms are omitted as they are completely standard. In this presentation, a static term s is either a static boolean term B of the sort $bool$, or a static integer I of the sort int, or a prop P of the sort $prop$, or a type T of the sort $type$. In practice, we allow the programmer to introduce new sorts through datasort declarations, which are rather similar to datatype declarations in ML. We assume some primitive functions c_B and c_I when forming static terms of the sorts $bool$ and int; for instance, we can form terms such as $I_1 + I_2$, $I_1 - I_2$, $I_1 \leq I_2$, $\neg B$, $B_1 \wedge B_2$, etc. We use $\overline{B}$ for a sequence of static boolean terms and $\Sigma; \overline{B} \models B$ for a constraint that means for any substitution $\Theta : \Sigma$, if each static boolean term in $\overline{B}[\Theta]$ equals *tt* then so does $B[\Theta]$. Note that we use $\Theta : \Sigma$ to mean $\emptyset \vdash \Theta(a) : \Sigma(a)$ holds for each $a \in \mathbf{dom}(\Theta) = \mathbf{dom}(\Sigma)$. In practice, such a constraint relation is often determined by some automatic decision procedure.

We now briefly explain some unfamiliar syntax of $\lambda_{pf}^{\forall,\exists}$.

- $B \supset T$ is called a guarded type and $B \wedge T$ is called an asserting type. As an example, the following type is for a function from natural numbers to negative integers:

  $$\forall a_1 : int.a_1 \geq 0 \supset \\ (\mathbf{int}(a_1) \rightarrow \exists a_2 : int.(a_2 < 0) \wedge \mathbf{int}(a_2))$$

  The guard $a_1 \geq 0$ indicates that the function can only be applied to an integer that is greater than or equal to 0; the assertion $a_2 < 0$ means that each integer returned by the function is negative.
- The markers $\supset^+(\cdot), \supset^-(\cdot), \wedge(\cdot), \forall^+(\cdot), \forall^-(\cdot), \exists(\cdot)$ are introduced to establish a lemma needed for conducting inductive reasoning on typing derivations. Please see [25] for further explanation on this issue.

In addition, we introduce two type constructors $\mathbf{bool}$ and $\mathbf{int}$; given a static boolean term B, $\mathbf{bool}(B)$ is the singleton type in which the only value is the truth value of B; similarly, given an integer I, $\mathbf{int}(I)$ is the singleton type in which the only value is the integer I.

$$
\begin{array}{llcl}
\text{sorts} & \sigma & ::= & bool \mid int \mid prop \mid type \\
\text{static contexts} & \Sigma & ::= & \emptyset \mid \Sigma, a : \sigma \\
\text{static bool. terms} & B & ::= & b \mid c_B(s_1, \ldots, s_n) \\
\text{static int. terms} & I & ::= & i \mid c_I(s_1, \ldots, s_n) \\
\text{props} & P & ::= & \ldots \mid B \supset P \mid \forall a : \sigma.P \mid B \wedge P \mid \exists a : \sigma.P \\
\text{types} & T & ::= & \ldots \mid a \mid \mathbf{bool}(B) \mid \mathbf{int}(I) \mid B \supset T \mid \forall a : \sigma.T \mid B \wedge T \mid \exists a : \sigma.T \\
\text{proof terms} & \underline{d} & ::= & \ldots \mid \supset^+(\underline{d}) \mid \supset^-(\underline{d}) \mid \forall^+(\underline{d}) \mid \forall^-(\underline{d}) \mid \wedge(\underline{d}) \mid \mathbf{let}\ \wedge(\underline{x}) = \underline{d}_1\ \mathbf{in}\ \underline{d}_2 \mid \exists(\underline{d}) \mid \mathbf{let}\ \exists(\underline{x}) = \underline{d}_1\ \mathbf{in}\ \underline{d}_2 \\
\text{dynamic terms} & d & ::= & \ldots \mid \mathbf{if}(d_1, d_2, d_3) \mid \mathbf{let}\ \wedge(\underline{x}) = \underline{d}\ \mathbf{in}\ d \mid \mathbf{let}\ \exists(\underline{x}) = \underline{d}\ \mathbf{in}\ d \mid \\
 & & & \supset^+(d) \mid \supset^-(d) \mid \forall^+(d) \mid \forall^-(d) \mid \wedge(d) \mid \mathbf{let}\ \wedge(x) = d_1\ \mathbf{in}\ d_2 \mid \exists(d) \mid \mathbf{let}\ \exists(x) = d_1\ \mathbf{in}\ d_2
\end{array}
$$

Figure 7. The syntax for $\lambda_{pf}^{\forall,\exists}$

$$\frac{}{\Pi; \Delta, xf : T \vdash xf : T}\ \textbf{(ty-var)}$$

$$\frac{\Pi \vdash \underline{d} : P \quad \Pi; \Delta \vdash d : T}{\Pi; \Delta \vdash \langle \underline{d}, d \rangle : P * T}\ \textbf{(ty-pr-tup-i)}$$

$$\frac{\begin{array}{c}\Pi; \Delta \vdash d_1 : P * T_1 \\ \Pi, \underline{x} : P; \Delta, x : T_1 \vdash d_2 : T_2\end{array}}{\Pi; \Delta \vdash \mathbf{let}\ \langle \underline{x}, x \rangle = d_1\ \mathbf{in}\ d_2 : T_2}\ \textbf{(ty-pr-tup-e)}$$

$$\frac{\Pi, \underline{x} : P; \Delta \vdash v : T}{\Pi; \Delta \vdash \mathbf{lam}\ \underline{x}.v : P \to T}\ \textbf{(ty-pr-lam)}$$

$$\frac{\Pi; \Delta \vdash d : P \to T \quad \Pi \vdash \underline{d} : P}{\Pi; \Delta \vdash \mathbf{app}(d, \underline{d}) : T}\ \textbf{(ty-pr-app)}$$

$$\frac{\Pi \vdash \underline{d} : P \quad \Pi, \underline{x} : P; \Delta \vdash d : T}{\Pi; \Delta \vdash \mathbf{let}\ \underline{x} = \underline{d}\ \mathbf{in}\ d : T}\ \textbf{(ty-pr-let)}$$

$$\frac{}{\Pi; \Delta \vdash \langle\rangle : \mathbf{1}}\ \textbf{(ty-unit)}$$

$$\frac{\Pi; \Delta \vdash d_1 : T_1 \quad \Pi; \Delta \vdash d_2 : T_2}{\Pi; \Delta \vdash \langle d_1, d_2 \rangle : T_1 * T_2}\ \textbf{(ty-tup)}$$

$$\frac{\Pi; \Delta \vdash d : T_1 * T_2}{\Pi; \Delta \vdash \mathbf{fst}(d) : T_1}\ \textbf{(ty-fst)}$$

$$\frac{\Pi; \Delta \vdash d : T_1 * T_2}{\Pi; \Delta \vdash \mathbf{snd}(d) : T_2}\ \textbf{(ty-snd)}$$

$$\frac{\Pi; \Delta, x : T_1 \vdash d : T_2}{\Pi; \Delta \vdash \mathbf{lam}\ x.d : T_1 \to T_2}\ \textbf{(ty-lam)}$$

$$\frac{\Pi; \Delta \vdash d_1 : T_1 \to T_2 \quad \Pi; \Delta \vdash d_2 : T_1}{\Pi; \Delta \vdash \mathbf{app}(d_1, d_2) : T_2}\ \textbf{(ty-app)}$$

$$\frac{\Pi; \Delta \vdash d_1 : T_1 \quad \Pi; \Delta, x : T_1 \vdash d_2 : T_2}{\Pi; \Delta \vdash \mathbf{let}\ x = d_1\ \mathbf{in}\ d_2 : T_2}\ \textbf{(ty-let)}$$

$$\frac{\Pi; \Delta, f : T \vdash d : T}{\Pi; \Delta \vdash \mathbf{fix}\ f.d : T}\ \textbf{(ty-fix)}$$

Figure 5. The rules for assigning types to dynamic terms in λ_{pf}

A judgment for assigning a prop to a proof term is now of the form $\Sigma; \overline{B}; \Pi \vdash \underline{d} : P$, and the rules in Figure 4 need to be properly modified. Intuitively, such a judgment means that $\Pi[\Theta] \vdash \underline{d}[\Theta] : P[\Theta]$ holds for any substitution $\Theta : \Sigma$ such that $B[\Theta]$ holds for each B in $\overline{B}$. Some additional rules for assigning props to proof terms are given in Figure 8.

$$
\begin{array}{rcl}
|xf| & = & xf \\
|\langle \underline{d}, d \rangle| & = & |d| \\
|\mathbf{let}\ \langle \underline{x}, x \rangle = d_1\ \mathbf{in}\ d_2| & = & \mathbf{let}\ x = |d_1|\ \mathbf{in}\ |d_2| \\
|\mathbf{lam}\ \underline{x}.v| & = & |v| \\
|\mathbf{app}(\underline{d}, d)| & = & |d| \\
|\mathbf{let}\ \underline{x} = \underline{d}\ \mathbf{in}\ d| & = & |d| \\
|\langle d_1, d_2 \rangle| & = & \langle |d_1|, |d_2| \rangle \\
|\mathbf{fst}(d)| & = & \mathbf{fst}(|d|) \\
|\mathbf{snd}(d)| & = & \mathbf{snd}(|d|) \\
|\mathbf{lam}\ x.d| & = & \mathbf{lam}\ x.|d| \\
|\mathbf{app}(d_1, d_2)| & = & \mathbf{app}(|d_1|, |d_2|) \\
|\mathbf{let}\ x = d_1\ \mathbf{in}\ d_2| & = & \mathbf{let}\ x = |d_1|\ \mathbf{in}\ |d_2| \\
|\mathbf{fix}\ f.d| & = & \mathbf{fix}\ f.|d|
\end{array}
$$

Figure 6. The erasure function

Similarly, a judgment for assigning a type to a dynamic term is now of the form $\Sigma; \overline{B}; \Pi; \Delta \vdash d : T$, and the rules in Figure 5 need to be modified properly. Some additional rules for assigning types to dynamic terms are given in Figure 9.

Following the development of $\mathcal{ATS}$, it is a standard routine to establish the type soundness of $\lambda_{pf}^{\forall,\exists}$. Then we can prove a theorem in $\lambda_{pf}^{\forall,\exists}$ that corresponds to Theorem 2.6. In practice, we also need to allow the use of recursion in constructing proof terms. It is clear that we cannot support unrestricted general recursion as it would otherwise allow the construction of proof terms that are not normalizing and thus invalidate Theorem 2.3, which plays a crucial rôle in establishing Theorem 2.5. Instead, we follow the work in [23], providing a means for the programmer to define terminating proof terms by supplying a form of metrics. This point will be made clear when we present some examples in the next section. Another issue in practice is the need for recursive props (dataprops) and recursive types (datatypes), which are not present in $\lambda_{pf}^{\forall,\exists}$ for the sake of brevity. It should be understood that recursive props and recursive types can be readily added into $\lambda_{pf}^{\forall,\exists}$ without difficulty.[3] We now use a simple example to illustrate some of these mentioned issues.

In Figure 10, we declare a prop constructor $\underline{\mathbf{MUL}}$, where the concrete syntax indicates that there are three (proof) value constructors associated with $\underline{\mathbf{MUL}}$, which are given the following

[3] As for the definition of a recursive prop, we require that the defined prop itself have no negative occurrences in the definition. Otherwise, a nonterminating proof term can be constructed without using fixed-point operator.

$$\frac{\Sigma; \overline{B}, B; \Pi \vdash \underline{d} : P}{\Sigma; \overline{B}; \Pi \vdash \supset^+(\underline{d}) : B \supset P} \textbf{ (pr-}\supset\textbf{+)}$$

$$\frac{\Sigma; \overline{B}; \Pi \vdash \underline{d} : B \supset P \quad \Sigma; \overline{B} \models B}{\Sigma; \overline{B}; \Pi \vdash \supset^-(\underline{d}) : P} \textbf{ (pr-}\supset\textbf{-)}$$

$$\frac{\Sigma, a : \sigma; \overline{B}; \Pi \vdash \underline{d} : P}{\Sigma; \overline{B}; \Pi \vdash \forall^+(\underline{d}) : \forall a : \sigma.P} \textbf{ (pr-}\forall\textbf{+)}$$

$$\frac{\Sigma; \overline{B}; \Pi \vdash \underline{d} : \forall a : \sigma.P \quad \Sigma \vdash s : \sigma}{\Sigma; \overline{B}; \Pi \vdash \forall^-(\underline{d}) : P[a \mapsto s]} \textbf{ (pr-}\forall\textbf{-)}$$

$$\frac{\Sigma; \overline{B} \models B \quad \Sigma; \overline{B}; \Pi \vdash \underline{d} : P}{\Sigma; \overline{B}; \Pi \vdash \wedge(\underline{d}) : B \wedge P} \textbf{ (pr-}\wedge\textbf{+)}$$

$$\frac{\Sigma; \overline{B}; \Pi \vdash \underline{d}_1 : B \wedge P_1 \quad \Sigma; \overline{B}, B; \Pi, \underline{x} : P_1 \vdash \underline{d}_2 : P_2}{\Sigma; \overline{B}; \Pi \vdash \textbf{let } \wedge(\underline{x}) = \underline{d}_1 \textbf{ in } \underline{d}_2 : P_2} \textbf{ (pr-}\wedge\textbf{-)}$$

$$\frac{\Sigma \vdash s : \sigma \quad \Sigma; \overline{B}; \Pi \vdash \underline{d} : P[a \mapsto s]}{\Sigma; \overline{B}; \Pi \vdash \exists(\underline{d}) : \exists a : \sigma.P} \textbf{ (pr-}\exists\textbf{+)}$$

$$\frac{\begin{array}{c}\Sigma; \overline{B}; \Pi \vdash \underline{d}_1 : \exists a : \sigma.P_1 \\ \Sigma, a : \sigma; \overline{B}; \Pi, \underline{x} : P_1 \vdash \underline{d}_2 : P_2\end{array}}{\Sigma; \overline{B}; \Pi \vdash \textbf{let } \exists(\underline{x}) = \underline{d}_1 \textbf{ in } \underline{d}_2 : P_2} \textbf{ (pr-}\exists\textbf{-)}$$

Figure 8. Some additional rules for assigning props to proof terms

constant props:

$$\begin{array}{rcl} \mathit{MULbas} & : & \forall n : \mathit{int}.() \rightarrow \underline{\mathbf{MUL}}(0, n, 0) \\ \mathit{MULind} & : & \forall m : \mathit{int}.\forall n : \mathit{int}.\ m > 0 \supset \\ & & (\underline{\mathbf{MUL}}(m-1, n, p-n) \rightarrow \underline{\mathbf{MUL}}(m, n, p)) \\ \mathit{MULneg} & : & \forall m : \mathit{int}.\forall n : \mathit{int}.\ m < 0 \supset \\ & & (\underline{\mathbf{MUL}}(-m, n, -p) \rightarrow \underline{\mathbf{MUL}}(m, n, p)) \end{array}$$

Given integers I_1, I_2, I_3, it is clear that $I_1 * I_2 = I_3$ holds if and only if $\underline{\mathbf{MUL}}(I_1, I_2, I_3)$ can be assigned to a closed (proof) value. In essence, $\mathit{MULbas}, \mathit{MULind}$ and *MULneg* correspond to the following three equations in an inductive definition of the multiplication function on integers:

$$\begin{array}{rcl} 0 * n & = & 0; \\ m * n & = & (m-1) * n + n \ \text{ if } m > 0; \\ m * n & = & -((-m) * n) \ \text{ if } m < 0. \end{array}$$

In Figure 10, lemma_1 is defined as a proof function of the following prop:

$$\begin{array}{l} \forall m : \mathit{nat}.\forall n : \mathit{nat}.\forall p : \mathit{int}. \\ \quad \underline{\mathbf{MUL}}(n, m, p) \rightarrow \underline{\mathbf{MUL}}(n, m+1, p+n) \end{array}$$

Note that we use the keyword `prfun` to declare a proof function. Essentially, lemma_1 represents an inductive proof of $n * m = p \supset n * (m+1) = p + n$ for all natural numbers m, n and integers p, where the induction is on n. In particular, the following two linear arithmetic constraints, which can be easily verified, are generated when the two clauses in the body of lemma_1 are type-checked:

$$\begin{array}{l} \forall n : \mathit{nat}.\forall p : \mathit{int}.\ n = 0 \supset (p = 0 \supset 0 = p + n) \\ \forall m : \mathit{nat}.\forall n : \mathit{nat}.\forall p : \mathit{int}.\forall n' : \mathit{int}.\forall p' : \mathit{int}. \\ \quad n = n' + 1 \supset (p = p' + m \supset p + n = (p' + n') + (m + 1)) \end{array}$$

However, in order for lemma_1 to represent a proof, we need to show that lemma_1 is a total function, that is, given *pf* of prop $\underline{\mathbf{MUL}}(I_2, I_1, I_3)$ for natural numbers I_1 and I_2 and integer I_3, $\mathit{lemma}_1(\mathit{pf})$ is guaranteed to return a proof value of prop

$$\frac{\begin{array}{c}\Sigma; \overline{B}; \Pi; \Delta \vdash d_1 : \mathbf{bool}(B) \\ \Sigma; \overline{B}, B; \Pi; \Delta \vdash d_2 : T \\ \Sigma; \overline{B}, \neg B; \Pi; \Delta \vdash d_3 : T\end{array}}{\Sigma; \overline{B}; \Pi; \Delta \vdash \textbf{if}(d_1, d_2, d_3) : T} \textbf{ (ty-if)}$$

$$\frac{\begin{array}{c}\Sigma; \overline{B}; \Pi \vdash \underline{d} : B \wedge P \\ \Sigma; \overline{B}, B; \Pi, \underline{x} : P; \Delta \vdash d : T\end{array}}{\Sigma; \overline{B}; \Pi; \Delta \vdash \textbf{let } \wedge(\underline{x}) = \underline{d} \textbf{ in } d : T} \textbf{ (ty-pr-}\wedge\textbf{-)}$$

$$\frac{\begin{array}{c}\Sigma; \overline{B}; \Pi \vdash \underline{d} : \exists a : \sigma.P \\ \Sigma, a : \sigma; \overline{B}; \Pi, \underline{x} : P; \Delta \vdash d : T\end{array}}{\Sigma; \overline{B}; \Pi; \Delta \vdash \textbf{let } \exists(\underline{x}) = \underline{d} \textbf{ in } d : T} \textbf{ (ty-pr-}\exists\textbf{-)}$$

$$\frac{\Sigma; \overline{B}, B; \Pi; \Delta \vdash d : T}{\Sigma; \overline{B}; \Pi; \Delta \vdash \supset^+(d) : B \supset T} \textbf{ (ty-}\supset\textbf{+)}$$

$$\frac{\Sigma; \overline{B}; \Pi; \Delta \vdash d : B \supset T \quad \Sigma; \overline{B} \models B}{\Sigma; \overline{B}; \Pi; \Delta \vdash \supset^-(d) : T} \textbf{ (ty-}\supset\textbf{-)}$$

$$\frac{\Sigma, a : \sigma; \overline{B}; \Pi; \Delta \vdash d : T}{\Sigma; \overline{B}; \Pi; \Delta \vdash \forall^+(d) : \forall a : \sigma.T} \textbf{ (ty-}\forall\textbf{+)}$$

$$\frac{\Sigma; \overline{B}; \Pi; \Delta \vdash d : \forall a : \sigma.T \quad \Sigma \vdash s : \sigma}{\Sigma; \overline{B}; \Pi; \Delta \vdash \forall^-(d) : T[a \mapsto s]} \textbf{ (ty-}\forall\textbf{-)}$$

$$\frac{\Sigma; \overline{B} \models B \quad \Sigma; \overline{B}; \Pi; \Delta \vdash d : T}{\Sigma; \overline{B}; \Pi; \Delta \vdash \wedge(d) : B \wedge T} \textbf{ (ty-}\wedge\textbf{+)}$$

$$\frac{\begin{array}{c}\Sigma; \overline{B}; \Pi; \Delta \vdash d_1 : B \wedge T_1 \\ \Sigma; \overline{B}, B; \Pi; \Delta, x : T_1 \vdash d_2 : T_2\end{array}}{\Sigma; \overline{B}; \Pi; \Delta \vdash \textbf{let } \wedge(x) = d_1 \textbf{ in } d_2 : T_2} \textbf{ (ty-}\wedge\textbf{-)}$$

$$\frac{\Sigma \vdash s : \sigma \quad \Sigma; \overline{B}; \Pi; \Delta \vdash d : T[a \mapsto s]}{\Sigma; \overline{B}; \Pi; \Delta \vdash \exists(d) : \exists a : \sigma.T} \textbf{ (ty-}\exists\textbf{+)}$$

$$\frac{\begin{array}{c}\Sigma; \overline{B}; \Pi; \Delta \vdash d_1 : \exists a : \sigma.T_1 \\ \Sigma, a : \sigma; \overline{B}; \Pi; \Delta, x : T_1 \vdash d_2 : T_2\end{array}}{\Sigma; \overline{B}; \Pi; \Delta \vdash \textbf{let } \exists(x) = d_1 \textbf{ in } d_2 : T_2} \textbf{ (ty-}\exists\textbf{-)}$$

Figure 9. Some additional rules for assigning types to dynamic terms

$\underline{\mathbf{MUL}}(I_2, I_1 + 1, I_3 + I_2)$. Generally speaking, when implementing a recursive proof function in $\mathcal{ATS}$, the programmer is required to provide a metric that can be used to verify the termination of the function. A thorough study on using such metrics for verifying program termination can be found in [22, 23]. In the definition of lemma_1, $\langle n \rangle$ is the provided metric for verifying that lemma_1 is terminating; when lemma_1 is applied to a value of prop $\underline{\mathbf{MUL}}(I_2, I_1, I_3)$, the label $\langle I_2 \rangle$ is associated with this call; in case a recursive call to lemma_1 is made subsequently, the label associated with the recursive call is $\langle I_2 - 1 \rangle$ (since pf' in the definition of lemma_1 is given the type $\underline{\mathbf{MUL}}(n-1, m, p-m)$), which is strictly less than the label $\langle I_2 \rangle$ associated with the original call; as a label associated with lemma_1 is always a natural number, it is evident that lemma_1 is terminating. To show that lemma_1 is total, we also need to verify that pattern matching in the definition of lemma_1 can never fail, which is a topic that is already studied elsewhere [20, 24].

```
dataprop MUL (int, int, int) =
  | {n:int} MULbas (0, n, 0)

  | {m:int,n:int,p:int | m > 0}
      MULind (m, n, p) of MUL (m-1, n, p-n)

  | {m:int,n:int,p:int | m > 0}
      MULneg (m, n, p) of MUL (~m, n, ~p)

(* <n> is a termination metric *)
prfun lemma1 {m:nat, n:nat, p:int} .<n>.
   (pf: MUL (n, m, p)): MUL (n, m+1, p+n) =
  case pf of MULbas =>
    MULbas | MULind pf' => MULind (lemma1 pf')
```

Figure 10. A dataprop for encoding integer multiplication

4. Examples

We present a few examples in ATS to give the reader some concrete feel as to how combining programming with theorem proving can be put into practice. There are also a large number of more realistic examples that can be found on-line [27].

Of course, we need a process to elaborate programs written in the concrete syntax of ATS into the (kind of) formal syntax of $\lambda_{pf}^{\forall,\exists}$. This is a rather involved process, and we unfortunately could not formally describe it in this paper and thus refer the interested reader to [26] for further details. Instead, we are to provide some (informal) explanation to facilitate the understanding of the concrete syntax we use.

We use ieq, $ipred$, $iadd$, $isub$ and $imul$ for the equality function, the predecessor function, the addition function, the subtraction function and the multiplication function on integers, which are given the following types:

$$\begin{array}{rcl} ieq & : & \forall a_1 : int. \forall a_2 : int. \\ & & (\mathbf{int}(a_1), \mathbf{int}(a_2)) \rightarrow \mathbf{bool}(a_1 = a_2) \\ ipred & : & \forall a : int.\ \mathbf{int}(a) \rightarrow \mathbf{int}(a - 1) \\ iadd & : & \forall a_1 : int. \forall a_2 : int. \\ & & (\mathbf{int}(a_1), \mathbf{int}(a_2)) \rightarrow \mathbf{int}(a_1 + a_2) \\ isub & : & \forall a_1 : int. \forall a_2 : int. \\ & & (\mathbf{int}(a_1), \mathbf{int}(a_2)) \rightarrow \mathbf{int}(a_1 - a_2) \\ imul & : & \forall a_1 : int. \forall a_2 : int. \\ & & (\mathbf{int}(a_1), \mathbf{int}(a_2)) \rightarrow \\ & & \exists a_3 : int.\ \underline{\mathbf{MUL}}(a_1, a_2, a_3) * \mathbf{int}(a_3) \end{array}$$

Note that $imul$ is *not* given the following type:

$$\forall m : int. \forall n : int.\ (\mathbf{int}(m), \mathbf{int}(n)) \rightarrow \mathbf{int}(m * n)$$

as $m * n$, which is nonlinear, is not allowed to be a type index in ATS.

4.1 List Concatenation

The code for implementing *concat* is given in Figure 11, which concatenates a given list of lists together. We write $'(\ldots)$ to form a tuple, where the quote symbol ($'$) is used solely for the purpose of parsing. Also, we use the bar symbol ($|$) as a separator to separate proofs from programs. When given an argument *xss* of type $\mathbf{list}(\mathbf{list}(T, I_2), I_1)$, the function *concat* returns a pair (pf, res) such that *pf* is a proof value of prop $\underline{\mathbf{MUL}}(I_1, I_2, I_3)$ for some integer I_3 and *res* is a list of type $\mathbf{list}(T, I_3)$. Therefore, *pf* acts as a witness to certify that the length of *res* is $I_1 * I_2$. Now suppose

```
fun concat {a:type, m:nat, n:nat}
   (xxs: list (list (a, n), m))
  : '(MUL (m, n, p) | list (a, p)) =
  case xxs of
    | nil => nil
    | cons (xs, xss) =>
      let val '(pf | res) = concat xss in
         '(MULind (pf) | append (xs, res))
      end
```

Figure 11. An implementation of the list concatenation function

```
fun fact1 {a:nat} (x: int a): Int =
  if x ieq 0 then 1 else
    let val '(pf | r) = x imul fact1 (ipred x) in
       r
    end

// <a1> is the termination metric
prfun lemma2 {a1:nat, a2:nat, a3:int} .<a1>.
  (pf: MUL (a1, a2, a3)): [a3 >= 0] '() =
  case pf of
    MULbas => '() | MULind pf => lemma2 pf

fun fact2 {a:nat} (x: int a): Nat =
  if x ieq 0 then 1 else
    let
       val '(pf | r) = x imul fact2 (ipred x)
       prval _ = lemma2 (pf) // proves r >= 0
    in
       r
    end

dataprop FACT (int, int) =
  | FACTbas (0, 1)
  | {n:int, r:int, r1:int | n > 0}
      FACTind(n,r) of (FACT(n-1,r1), MUL(n,r1,r))

fun fact3 {a:nat} (x: int a)
  : [r:int] '(FACT (a, r) | int r) =
  if x ieq 0 then '(FACTbas | 1) else
    let
       val '(pf1 | r1) = fact3 (ipred x)
       val '(pf2 | r) = x imul r1
    in
       '(FACTind (pf1, pf2) | r)
    end
```

Figure 12. Some implementations of the factorial function

we would like to assign *concat* the following type:

$$\begin{array}{l} \forall a : type. \forall m : nat. \forall n : nat. \\ \quad \mathbf{list}(\mathbf{list}(a, n), m) \rightarrow \exists p : int.\ \underline{\mathbf{MUL}}(n, m, p) * \mathbf{list}(a, p) \end{array}$$

which is obtained from replacing the prop $\underline{\mathbf{MUL}}(m, n, p)$ with the prop $\underline{\mathbf{MUL}}(n, m, p)$ in the above type assigned to *concat*. Then we need to replace $MULind(pf)$ in the definition of *concat* with $lemma_1(pf)$, where $lemma_1$ is defined in Figure 10.

4.2 Different Implementations of the Factorial Function

We present three implementations of the factorial function in Figure 12 to make an interesting point. The function $fact_1$ is given the following type:

$$\forall a : nat.\ \mathbf{int}(a) \rightarrow \mathbf{Int}$$

where $\mathbf{Int}$, defined to be $\exists a : int.\ \mathbf{int}(a)$, is the type for all integers. This type simply means that $fact_1$ is a function from natural numbers to integers. Note that we use the bar symbol (|) in the concrete syntax to separate proof terms from dynamic terms in a tuple.

The function $fact_2$ is given the following type:

$$\forall a : nat.\ \mathbf{int}(a) \to \mathbf{Nat}$$

where $\mathbf{Nat}$, defined to be $\exists a : nat.\mathbf{int}(a)$, is the type for all natural numbers. Hence, $fact_2$ is a function from natural numbers to natural numbers. When implementing $fact_2$, we need to prove that the product of two natural numbers is a natural number. This is done by the proof function $lemma_2$ defined in Figure 12, which is assigned the following prop:

$$\begin{array}{c}\forall a_1 : nat.\forall a_2 : nat.\forall a_3 : int. \\ \underline{\mathbf{MUL}}(a_1, a_2, a_3) \to (a_3 \geq 0) \wedge \mathbf{1}\end{array}$$

where $\mathbf{1}$ is the unit prop. The syntax `.<a1>.` in the header of the definition of $lemma_2$ indicates that a_1 is the metric (provided by the programmer) for establishing the termination of $lemma_2$. Note that the keyword `prval` in the body of the function $fact_2$ indicates pattern matching on proof values, which is erased before program execution.

Next we declare a prop constructor $\underline{\mathbf{FACT}}$, which forms a prop $\underline{\mathbf{FACT}}(I_1, I_2)$ when applied to two given integers I_1 and I_2; there is a closed value of prop $\underline{\mathbf{FACT}}(I_1, I_2)$ if and only if I_2 equals the factorial of I_1. The function $fact_3$ is assigned the following type:

$$\forall a_1 : nat.\ \mathbf{int}(a_1) \to \exists a_2 : int.\underline{\mathbf{FACT}}(a_1, a_2) * \mathbf{int}(a_2)$$

When applied to a value of type $\mathbf{int}(I_1)$ for some natural number I_1, $fact_3$ returns a proof *pf* of prop $\underline{\mathbf{FACT}}(I_1, I_2)$ for some integer I_2 and a value of type $\mathbf{int}(I_2)$. Of course, the proof is not actually constructed at run-time.

Clearly, $fact_1, fact_2, fact_3$ all implement the factorial function, but the types assigned to them become more and more accurate. Intuitively speaking, the programmer is given some freedom in ATS to determine the extent of theorem proving to be involved based on the invariants that need to be captured.

4.3 Implementing the call-by-value evaluation for the pure untyped λ-calculus

In contrast to extracting programs out of proofs, we emphasize that dynamic functions such as $fact_1$, $fact_2$ and $fact_3$ are not meant to correspond to any proofs in the first place. In particular, they are not assumed to be terminating (though $fact_1, fact_2, fact_3$ are all terminating) and may incur effects (e.g., updating references, raising exceptions). This is crucial to practical programming. To further stress this point, we give an implementation of the call-by-value evaluation for the pure untyped λ-calculus in Figure 13.

We declare a datasort tm such that each static term of the sort tm represents an untyped λ-term. For instance, the static term $\mathit{TMlam}(\lambda x : tm.\mathit{TMlam}(\lambda y : tm.\mathit{TMapp}(y, x)))$ represents $\lambda x.\lambda y.y(x)$. This representation strategy is referred to as higher-order abstract syntax [16]. We then declare a prop constructor $\underline{\mathbf{EVAL}}$ that forms a prop $\underline{\mathbf{EVAL}}(s_1, s_2)$ when applied to two static terms s_1 and s_2 of the sort tm; there exists a closed proof value of prop $\underline{\mathbf{EVAL}}(s_1, s_2)$ if and only if the λ-term represented by s_1 evaluates to the λ-term represented by s_2. We then declare a type constructor $\mathbf{EXP}$ which forms a type $\mathbf{EXP}(s)$ for each term s of the sort tm such that an untyped λ-term represented by s can be represented by a dynamic value of type $\mathbf{EXP}(s)$.[4] We next implement a function $evaluate$ of the following type:

```
datasort tm =
  TMlam of (tm -> tm) | TMapp of (tm, tm)

dataprop EVAL (tm, tm) =
  | {f: tm -> tm} EVALlam (TMlam f, TMlam f)
  | {t1: tm, t2: tm, f1: tm -> tm, v2: tm, v: tm}
      EVALapp (TMapp (t1, t2), v) of
        (EVAL (t1, TMlam f1),
         EVAL (t2, v2),
         EVAL (f1 v2, v))

datatype EXP (tm) =
  | {f: tm -> tm}
      EXPlam(TMlam f) of {t:tm} EXP t -> EXP(f t)
  | {t1: tm, t2: tm}
      EXPapp(TMapp (t1, t2)) of (EXP t1, EXP t2)

fun evaluate {t: tm} (e: EXP t)
  : [t': tm] '(EVAL (t, t') | EXP t') =
  case e of
    | EXPlam _ => '(EVALlam | e)
    | EXPapp (e1, e2) =>
        let
            val '(pf1 | EXPlam f1) = evaluate e1
            val '(pf2 | v2) = evaluate e2
            val '(pf3 | v) = evaluate (f1 v2)
        in
            '(EVALapp (pf1, pf2, pf3) | v)
        end
```

Figure 13. An implementation of the call-by-value evaluation for the pure untyped λ-calculus

$$\forall a : tm.\mathbf{EXP}(a) \to \exists a' : tm.\underline{\mathbf{EVAL}}(a, a') * \mathbf{EXP}(a')$$

When applied to a value of type $\mathbf{EXP}(s)$ for some static term s of the sort tm, $evaluate$ can only return a value of type $\mathbf{EXP}(s')$ such that a proof value of prop $\underline{\mathbf{EVAL}}(s, s')$ exists. However, it may never return as there certainly exist λ-terms that do not evaluate to any values. In other words, $evaluate$ is not a total function and thus cannot in principle be extracted out of any (valid) proof.

4.4 Safe Matrix Subscripting

We now present a simple but realistic example. In Figure 14, we first implement a proof function $lemma_3$. We use the keyword `prfun` in the concrete syntax to indicate that a proof function is defined. Essentially, $lemma_3$ proves the statement that $i * col + col \leq row * col$ holds if col, row and i are natural numbers and $i < row$ holds.

In the definition of $lemma_3$, the syntax `.<row>.` indicates that row is a metric supplied by the programmer; it can be easily verified that the metric decreases when a recursive call to $lemma_3$ is made in the body of $lemma_3$; this guarantees $lemma_3$ is terminating [23]. Also, it can be easily verified that pattern matching in the body of $lemma_3$ is exhaustive, and thus $lemma_3$ is a total function.

We next show in Figure 14 how a safe matrix subscripting function *matrixSub* is implemented. Given a type T and an integer I, we can form a type $\mathbf{array}(T, I)$ in ATS for arrays of size I

[4] This is a higher-order representation that is not adequate as there are dynamic values of type $\mathbf{EXP}(s)$ that do not correspond to the λ-term represented by s. In Appendix, we are to present a more realistic implementation of the call-by-value evaluation of λ-calculus, which makes use of a first-order adequate representation for λ-terms and avoids the need for substitution by forming closures.

```
prfun lemma3
     {row:nat, col:nat, size:int, i:nat, p:int |
      i < row} .<row>.
     (pf1: MUL(row,col,size), pf2: MUL(i,col,p))
    : [p + col <= size] unit =
  case pf1 of
    | MULind (pf1) => begin
      case pf2 of
        | MULbas =>
          let val _ = lemma2 (pf1) in '() end

        | MULind pf2 =>
          let val _ = lemma3 (pf1, pf2) in
             '()
          end
      end

typedef matrix (a: type, row: int, col: int) =
  [size:int]
    '(MUL (row,col,size) |
      int row, int col, array (a, size))

fun matrixSub
   {a:type, row:nat, col:nat, i:nat, j:nat |
    i < row, j < col}
   (M: matrix(a,row,col), i: int i, j: int j): a =
  let
     val '(pf1 | row, col, A) = M
     val '(pf2 | p) = i imul col

     // proves: p >= 0
     prval _ = lemma2 (pf2)

     // proves: p + col <= row * col
     prval _ = lemma3 (pf1, pf2)
  in
     arraySub (A, p iadd j)
  end
```

Figure 14. An example of combining programs with proofs

in which each element is of type T. The usual array subscripting function *arraySub* is given the following type:

$$\forall a : type.\forall n : nat.\forall i : nat.$$
$$i < n \supset ((\mathbf{array}(a, n), \mathbf{int}(i)) \rightarrow a)$$

Therefore, the index used to access an array is required to be within the bounds of the array (we assume the index of a given array ranges from 0 until $n - 1$, where n is the size of the array). In Figure 14, we define a type constructor **matrix**; given a type T and two integers I_1 and I_2, $\mathbf{matrix}(T, I_1, I_2)$ is defined to be:

$$\exists p : int.\ \underline{\mathbf{MUL}}(I_1, I_2, p) * \mathbf{int}(I_1) * \mathbf{int}(I_2) * \mathbf{array}(T, p)$$

which indicates that a matrix of dimension I_1 by I_2 is represented as an array of size $I_1 * I_2$. The function *matrixSub* implemented in Figure 14 is assigned the following type as can be expected:

$$\forall a : type.\forall row : nat.\forall col : nat.\forall i : nat.\forall j : nat.$$
$$i < row \supset (j < col \supset$$
$$((\mathbf{matrix}(a, row, col), \mathbf{int}(i), \mathbf{int}(j)) \rightarrow a))$$

The following two lines of code in the definition of *matrixSub* are uncommon, and we now provide some explanation.

```
prval _ = lemma2 (pf2) // ...
prval _ = lemma3 (pf1, pf2) // ...
```

The use of the keyword `prval` is to indicate that the pattern following it is to match a proof value. If we assume that pf_2 is of prop $\underline{\mathbf{MUL}}(i, col, p)$ for some integer p, then $lemma_2(pf_2)$ is of prop $(p \geq 0) \wedge \mathbf{1}$; the code `prval _ = lemma2 (pf2)` essentially elaborates into **let** $\wedge(\underline{x}) = lemma_2(pf_2)$ **in** ..., which means $p \geq 0$ can be assumed when we solve the constraints generated in the scope of this let-binding; the typing rule involved here is **(ty-pr-$\wedge$-)**. Similarly, `prval _ = lemma3 (pf1, pf2)` essentially elaborates into **let** $\wedge(\underline{x}) = lemma_3(pf_1, pf_2)$ **in** ...; the prop of $lemma_3(pf_1, pf_2)$ is $(p + col \leq size) \wedge \mathbf{1}$, where we assume pf_1 is of prop $\underline{\mathbf{MUL}}(row, col, size)$, and thus $p + col \leq size$ can be assumed when we solve the constraints generated in the scope of this let-binding. Lastly, we need to show that both $p + j \geq 0$ and $p + j < size$ hold when calling $arraySub(A, p\ iadd\ j)$; the former constraint is easily proven since j is a natural number and $p \geq 0$ can be assumed; the latter constraint is also easily proven since both $p + col \leq size$ and $j < col$ can be assumed.

Hence, if the pair of supplied indexes i and j are natural numbers satisfying $i < row$ and $j < col$, accessing a matrix of dimension row by col via *matrixSub* is guaranteed to be safe. What is significant here is that this property is captured in the type system of ATS.

After matrix subscripting is properly handled, it is straightforward to implement various other functions (e.g., multiplication) on matrices. At this point, we stress that the programmer may also decide to insert run-time array bound checks to implement the matrix subscripting function *matrixSub*. By doing so, it is no longer necessary to construct the proof function $lemma_3$, though this also means that the absence of illegal array subscripting can no longer be guaranteed in the underlying type system.

5. Related Work and Conclusion

A fundamental problem in programming is to find approaches that can effectively facilitate the construction of safe and reliable software, and we have so far seen numerous attempts to address this problem. An interesting idea is to build a language based on Martin-Löf's constructive type theory [9, 13] or its variants in which software specifications can be formally stated and proven and an implementation can be algorithmically extracted from the proof of a specification to guarantee that the extracted implementation meets the specification. While the practicality of such an idealistic approach to software development is yet to be proven, the notion expressed in this approach of integrating software design and implementation in a verifiably consistent manner is certainly inspiring.

Constructive type theory, which was originally proposed by Martin-Löf primarily for the purpose of establishing a foundation for mathematics, requires pure reasoning on programs (or proofs, more precisely). This requirement seems to have imposed a fundamental limitation on the use of constructive type theory in practical programming as pure reasoning on (large and realistic) programs seems simply untenable. We have recently rectified the situation by formalizing a framework *Applied Type System* ($\mathcal{ATS}$) [25, 27], completely eliminating the need for pure reasoning on programs. Also, by introducing the notion of conditional type equality, we have made $\mathcal{ATS}$ highly expressive in capturing programming invariants. For instance, we have already formally demonstrated that $\mathcal{ATS}$ can be used as a basis to support in a typeful manner a variety of programming paradigms such as functional programming, imperative programming, object-oriented programming, modular programming meta-programming, etc.

In $\mathcal{ATS}$, a constraint relation is involved in determining type equality. In order to support effective constraint solving, we adopted a design in the past that imposes certain restrictions on the syntactic form of constraints that are allowed in practice. For instance, arithmetic constraints are required to be linear in the current implementation of ATS [27]. While this is a simple design, it is evidently *ad hoc* by its nature and can also be too restrictive, sometimes, in a situation where nonlinear constraints need to be handled. We have presented a different design in this paper. Instead of imposing restrictions, we provide a means for the programmer to construct proofs that attest to the validity of constraints.

It is interesting to see a comparison between our design for combining programs with proofs and theorem proving systems such as NuPrl [4] (based on Martin-Löf's constructive type theory) and Coq [7] (based on the calculus of construction [5]). In order to reason effectively about program properties within a type theory, the underlying functional language of a theorem proving system such as Coq or NuPrl is often required to be pure, making it difficult to support many realistic programming features (e.g., general recursion, reference, exception). In general, programming in such a setting amounts to constructing proofs (of specifications) and programs are automatically extracted out of the constructed proofs. This means that only total programs can be constructed in principle. Therefore, such a programming paradigm can often be inflexible or even infeasible in many common situations. For instance, partial functions such as $evaluate$ mentioned in Section 4 are rather common in practice and they in principle cannot be extracted from any proofs. To a large extent, this argument also applies to Epigram [11], a recently developed functional programming language with a dependent type system based on UTT [8], and it is yet to be seen whether monads can be successfully employed to incorporate effects (including partiality of functions) into Epigram in support of practical programming.

The sorts $prop$ and $type$ in ATS roughly correspond to the kinds $Prop$ and Set in Coq [14]. However, there is a subtle difference. In λ_{pf}, types may contain props but props may never contain types. For instance, $P * T$ is a type and can never be a prop. On the other hand, terms of kind $Prop$ may contain terms of kind Set in Coq and vice versa. This difference is profound as it makes it possible to construct in ATS programs that may be nonterminating, raising exceptions or causing effects, which on the other hand is forbidden in Coq. Moreover, the design we have presented for combining programs with proofs is largely rooted in a programming language, which can readily accommodate programming features such as general recursion and effects. Intuitively speaking, the design somewhat provides the programmer with some flexibility in determining the extent of theorem proving to be involved according to the invariants that need to be captured. This design is fundamentally different from program extraction.

The theme of combining programs with proofs is also proposed in the design of the programming language Ωemga [19]. The type system of Ωemga is largely built on top of a notion called *equality constrained types* (a.k.a. phantom types [3]), which are closely related to the notion of guarded recursive datatypes [28]. In Ωemga, there seems no strict separation between programs and proofs. In particular, proofs need to be constructed at run-time, and thus the serious problem with proof construction at run-time as is mentioned in Section 1 does occur in Ωemga. Also, an approach to simulating dependent types through the use of type classes in Haskell is given in [10], which is casually related to proof construction in our design. However, this approach does not address the issue of proof erasure, which on the other hand is key in our design. Furthermore, there is currently no facility in Haskell for verifying the totality of a function and a proof function such as $lemma_1$ cannot really be adequately simulated. Please see [2] for a critique on the practicality of simulating dependent types in Haskell.

Another line of related work is the formation of a type system in support of certified binaries [18], in which the idea of a complete separation between types and programs is also employed. Basically, the notions of type language and computational language in the type system correspond to the notions of statics and dynamics in $\mathcal{ATS}$, respectively, though the type language is based on the calculus of constructions extended with inductive definitions (CiC) [15]. However, the notion of a constraint relation in $\mathcal{ATS}$ does not have a counterpart in [18]. Instead, the equality between two types is determined by comparing the normal forms of these types. Also, there seems so far no attempt to build a source programming language, and in particular, the theme of combining programs with proofs (which is done by the programmer) is not addressed there.

In summary, we have presented a novel design in this paper for combining programs with proofs in support of the use of (advanced) types in capturing program invariants, opening a promising avenue to making theorem proving available for practical programming. In support of the practicality of this design, we have finished a running implementation of ATS and tested a variety of examples. In particular, a large part of the library of ATS involves the combination of programs and proofs as is described here. The presented design to support programming with theorem proving is both general and flexible, and we naturally expect to capture more program properties by exploring other logics (in addition to intuitionistic logic) and proof systems and investigating whether they can be incorporated into ATS. As a matter of fact, we have already succeeded in developing a proof system (based on a form of linear logic) to reason about properties on memory and then incorporated it into ATS [29] by following the design of combining programming with theorem proving. In the future, we plan to study whether reasoning about concurrency and distribution can also be supported in a similar fashion.

References

[1] CHEN, C., AND XI, H. Combining Programming with Theorem Proving, November 2004. Available at: `http://www.cs.bu.edu/~hwxi/ATS/PAPER/CPwTP.ps`.

[2] CHEN, C., ZHU, D., AND XI, H. Implementing Cut Elimination: A Case Study of Simulating Dependent Types in Haskell. In *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages* (Dallas, TX, June 2004), Springer-Verlag LNCS vol. 3057.

[3] CHENEY, J., AND HINZE, R. Phantom Types. Technical Report CUCIS-TR2003-1901, Cornell University, 2003. Available at `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cis/TR2003-1901`.

[4] CONSTABLE, R. L., ET AL. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[5] COQUAND, T., AND HUET, G. The calculus of constructions. *Information and Computation 76*, 2–3 (February–March 1988), 95–120.

[6] DANTZIG, G., AND EAVES, B. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A) 14* (1973), 288–297.

[7] DOWEK, G., FELTY, A., HERBELIN, H., HUET, G., MURTHY, C., PARENT, C., PAULIN-MOHRING, C., AND WERNER, B. The Coq proof assistant user's guide. Rapport Technique 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[8] LOU, Z. A unifying theory of dependent types: the schematic approach. Technical Report LFCS-92-202, University of Edinburgh, 1991.

[9] MARTIN-LÖF, P. *Intuitionistic Type Theory*. Bibliopolis, Naples, Italy, 1984.

[10] MCBRIDE, C. Faking It. *Journal of Functional Programming 12*, 4 & 5 (July 2002), 375–392.

[11] MCBRIDE, C., AND MCKINNA, J. The view from the left. *Journal of Functional Programming 14*, 1 (2004), 69–111.

[12] MILNER, R., TOFTE, M., HARPER, R. W., AND MACQUEEN, D.

The Definition of Standard ML (Revised). MIT Press, Cambridge, Massachusetts, 1997.

[13] NORDSTRÖM, B., PETERSSON, K., AND SMITH, J. M. *Programming in Martin-Löf's Type Theory*, vol. 7 of *International Series of Monographs on Computer Science*. Clarendon Press, Oxford, 1990.

[14] PAULIN-MOHRING, C. *Extraction de programmes dans le Calcul des Constructions*. Thèse de doctorat d'état, Université de Paris VII, Paris, France, 1989.

[15] PAULIN-MOHRING, C. Inductive Definitions in the System Coq: Rules and Properties. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications* (Utrecht, The Netherlands, 1993), M. Bezem and J. de Groote, Eds., vol. 664 of *Lecture Notes in Computer Science*, pp. 328–345.

[16] PFENNING, F., AND ELLIOTT, C. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation* (Atlanta, Georgia, June 1988), pp. 199–208.

[17] REYNOLDS, J. Separation Logic: a logic for shared mutable data structures. In *Proceedings of 17th IEEE Symposium on Logic in Computer Science (LICS '02)* (2002).

[18] SHAO, Z., SAHA, B., TRIFONOV, V., AND PAPASPYROU, N. A Type System for Certified Binaries. In *Proceedings of 29th Annual ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '02)* (Portland, OR, January 2002), pp. 217–232.

[19] SHEARD, T. Languages of the future. In *Proceedings of the Onward! Track of Objected-Oriented Programming Systems, Languages, Applications (OOPSLA)* (Vancouver, BC, October 2004), ACM Press.

[20] XI, H. Dead code elimination through dependent types. In *The First International Workshop on Practical Aspects of Declarative Languages* (San Antonio, January 1999), Springer-Verlag LNCS vol. 1551.

[21] XI, H. Dependent ML. Available at `http://www.cs.bu.edu/~hwxi/DML/DML.html`, 2001.

[22] XI, H. Dependent Types for Program Termination Verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science* (Boston, June 2001), pp. 231–242.

[23] XI, H. Dependent Types for Program Termination Verification. *Journal of Higher-Order and Symbolic Computation 15*, 1 (March 2002), 91–132.

[24] XI, H. Dependently Typed Pattern Matching. *Journal of Universal Computer Science 9*, 8 (2003), 851–872.

[25] XI, H. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003* (2004), Springer-Verlag LNCS 3085, pp. 394–408.

[26] XI, H. Dependent Types for Practical Programming via Applied Type System, September 2004. Available at `http://www.cs.bu.edu/~hwxi/academic/drafts/ATSdml.ps`

[27] XI, H. Applied Type System, 2005. Available at: `http://www.cs.bu.edu/~hwxi/ATS`.

[28] XI, H., CHEN, C., AND CHEN, G. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages* (New Orleans, LA, January 2003), ACM press, pp. 224–235.

[29] XI, H., AND ZHU, D. Views, Types and Viewtypes, October 2004. Available at: `http://www.cs.bu.edu/~hwxi/ATS/PAPER/VsTsVTs.ps`.

[30] XI, H., ZHU, D., AND LI, Y. Applied Type System with Stateful Views. Technical Report BUCS-2005-03, Boston University, 2005. Available at: `http://www.cs.bu.edu/~hwxi/ATS/PAPER/ATSwSV.ps`.

A. Implementing the call-by-value evaluation for the pure untyped λ-calculus via closures

We present an implementation of the call-by-value evaluation for the pure untyped λ-calculus in Figure 15. In this case, we use a first-order representation for λ-terms that essentially uses deBrujin indexes to represent free variables. Given a static term s of the sort tm, $\mathbf{EXP}_0(s)$ is the type for the dynamic value that represents the λ-term represented by s, and $\mathbf{VAL}(s)$ is the type for the closures that represents the λ-term represented by s. The following type is assigned to the function $evaluate$:

$$\forall a : tm.\, \mathbf{EXP}_0(a) \rightarrow \exists a' : tm.\, \underline{\mathbf{EVAL}}_0(a, a') * \mathbf{VAL}(a')$$

where $\underline{\mathbf{EVAL}}_0$ is like $\underline{\mathbf{EVAL}}$ in Section 4.

```
datasort tm = TMlam of (tm -> tm) | TMapp of (tm, tm)

datasort tms = TMSemp | TMSmore of (tms, tm)

dataprop EVAL (tm, tm, int) = // the third index is needed for form metrics
  | {f: tm -> tm} EVALlam (TMlam f, TMlam f, 0)
  | {t1: tm, t2: tm, f: tm -> tm, v1: tm, v2: tm, n1:nat, n2:nat, n3:nat}
      EVALapp (TMapp (t1, t2), v2, n1+n2+n3+1) of
        (EVAL (t1, TMlam f, n1), EVAL (t2, v1, n2), EVAL (f v1, v2, n3))

propdef EVAL0 (t:tm, t':tm) = [n:nat] EVAL (t, t', n)

dataprop ISVAL (tm) = {f: tm -> tm} ISVALlam (TMlam f) // a prop definition

prfun lemma1 {t:tm} (pf: ISVAL (t)): EVAL0 (t, t) = // a value evaluates to itself
  case pf of ISVALlam => EVALlam

// a lambda-term can only be evaluated to a value
prfun lemma2 {t: tm, t':tm, n:nat} .<n>. (pf: EVAL (t, t', n)): ISVAL (t') =
  case pf of
    | EVALlam => ISVALlam
    | EVALapp (_, _, pf3) => lemma2 pf3

datatype IN (tm, tms) = // deBruijn indexes
  | {ts: tms, t: tm} INone (t, TMSmore (ts, t))
  | {ts: tms, t: tm, t':tm} INshi (t, TMSmore (ts, t')) of IN (t, ts)

datatype EXP (tms, tm) = // first-order representation for lambda-terms
  | {ts: tms, t: tm} EXPvar (ts, t) of IN (t, ts)
  | {ts: tms, f: tm -> tm} EXPlam (ts, TMlam f) of {t: tm} EXP (TMSmore (ts, t), f t)
  | {ts: tms, t1: tm, t2: tm} EXPapp (ts, TMapp (t1, t2)) of (EXP (ts, t1), EXP (ts, t2))

typedef EXP0 (t: tm) = EXP (TMSemp, t) // a type definition

datatype VAL (tm) = // value representation
  | {ts: tms, f: tm -> tm } VALclo (TMlam f) of (ENV (ts), {t: tm} EXP (TMSmore (ts, t), f t))

and ENV (tms) = // environment representation
  | ENVnil (TMSemp)
  | {ts: tms, t: tm} ENVcons (TMSmore (ts, t)) of (ISVAL t | VAL (t), ENV (ts))

fun eval {ts: tms, t: tm} (env: ENV (ts), e: EXP (ts, t)): [t':tm] '(EVAL0 (t, t') | VAL (t')) =
  case e of
    | EXPvar i => evalVar (env, i)
    | EXPlam body => '(EVALlam | VALclo (env, body))
    | EXPapp (e1, e2) =>
        let
           val '(pf1 | VALclo (env', body)) = eval (env, e1)
           val '(pf2 | arg) = eval (env, e2)
           val '(pf3 | v) = eval (ENVcons (lemma2 pf2 | arg, env'), body)
        in
           '(EVALapp (pf1, pf2, pf3) | v)
        end

and evalVar {ts: tms, t: tm} (env: ENV (ts), i: IN (t, ts)): '(EVAL0 (t, t) | VAL (t)) =
  case i of
    | INone => let val ENVcons (pf | v, _) = env in '(lemma1 pf | v) end
    | INshi i => let val ENVcons (_ | _, env) = env in evalVar (env, i) end

fun evaluate {t: tm} (e: EXP0 (t))
  : [t':tm] '(EVAL0 (t, t') | VAL (t')) = eval (ENVnil, e)
```

Figure 15. An implementation of the call-by-value evaluation for the pure untyped λ-calculus via closures