Attributive Types for Proof Erasure *

Hongwei Xi

Boston University

Abstract. Proof erasure plays an essential role in the paradigm of programming with theorem proving. In this paper, we introduce a form of *attributive types* that carry an attribute to determine whether expressions assigned such types are eligible for erasure before run-time. We formalize a type system to support this form of attributive types and then establish its soundness. In addition, we outline an extension of the developed type system with dependent types and present some examples to illustrate its use in practice.

1 Introduction

In DML [Xi07,XP99], a restricted form of dependent types are introduced to allow for specification and inference of significantly more accurate type information (when compared to the types in ML) and thus further facilitate effective program error detection and compiler optimization through types. In contrast to the standard full dependent types (as in Martin-Löf's constructive theory [Mar84,NPS90]), types in DML can only depend on indexes drawn from a chosen index language, and type-checking a sufficiently annotated program in DML can be reduced to solving constraints from the chosen index language. This design makes it particularly straightforward to support common realistic programming features such as general recursion and effects (e.g., exceptions and references) in the presence of dependent types.

In order to solve constraints in an algorithmically effective manner, certain restrictions need to be imposed on indexes. For instance, constraints on integer indexes in DML are required to be linear,[1] and a constraint solver based on the Fourier-Motzkin variable elimination method [DE73] is then employed to solve such constraints. While this is indeed a simple design, it is inherently *ad hoc* and cannot handle a situation where nonlinear constraints (e.g., $\forall n : int.\ n * n \geq 0$) are involved. Let us now see a simple example that clarifies this point.

Let **list** be a type constructor that takes a type T and an integer I to form a type $\mathbf{list}(T, I)$ for lists of length I in which each element is of type T. The two list constructors associated with **list** are assigned the following types:

$$\begin{array}{rcl} nil & : & \forall\alpha.\ \mathbf{list}(\alpha, 0) \\ cons & : & \forall\alpha.\forall\iota.\ \iota \geq 0 \supset (\alpha * \mathbf{list}(\alpha, \iota) \rightarrow \mathbf{list}(\alpha, \iota + 1)) \end{array}$$

* This work is partially supported by NSF grants no. CCR-0229480 and no. CCF-0702665.

[1] More precisely, it is required that constraints on integer indexes in DML be converted into linear integer programming problems.

which indicate that *nil* forms a list of length 0 and *cons* takes an element and a list of length I to form a list of length $I+1$. We use α and ι for bound variables ranging over types and integers, respectively. Now assume that the function @ (infix) for appending two lists is given the following type:

$$\forall\alpha.\forall\iota_1.\forall\iota_2.\ \mathbf{list}(\alpha,\iota_1) * \mathbf{list}(\alpha,\iota_2) \rightarrow \mathbf{list}(\alpha,\iota_1+\iota_2)$$

In other words, appending two lists of length I_1 and I_2 yields a list of length $I_1 + I_2$. Naturally, the function that concatenates a list of length I_1 in which each element is a list of length I_2 is expected to have the following type:

$$\forall\alpha.\forall\iota_1.\forall\iota_2.\ \mathbf{list}(\mathbf{list}(\alpha,\iota_2),\iota_1) \rightarrow \mathbf{list}(\alpha,\iota_1 * \iota_2)$$

Unfortunately, this type is not allowed in DML as accepting nonlinear terms like $\iota_1 * \iota_2$ as type indexes would readily make constraint solving undecidable (or worse, intractable).

```
dataprop MUL (int, int, int) = // a prop for encoding multiplication
  | {n:int} MULbas (0, n, 0)
  | {m, n, p:int | m >= 0} MULind (m+1, n, p+n) of MUL (m, n, p)
  | {m, n, p:int | m > 0} MULneg (~m, n, ~p) of MUL (m, n, p)
```

Fig. 1. A dataprop for encoding multiplication on integers

To address this limitation, a fundamentally different design is adopted in ATS (which supersedes DML) to accommodate a paradigm that combines programming with theorem proving [CX05]. With this design, the programmer is given a means to handle nonlinear constraints by constructing explicit proofs attesting to the validity of such constraints (while linear constraints are still handled by an automatic constraint solver). In Figure 1, we declare a prop (i.e., type for proofs) 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 constant props (or c-props for short):

$$\begin{array}{lcl}
MULbas & : & \forall\iota.\ \underline{\mathbf{MUL}}(0,\iota,0) \\
MULind & : & \forall\iota_1.\forall\iota_2.\forall\iota_3.\ \iota_1 \geq 0 \supset (\underline{\mathbf{MUL}}(\iota_1,\iota_2,\iota_3) \rightarrow \underline{\mathbf{MUL}}(\iota_1+1,\iota_2,\iota_3+\iota_2)) \\
MULneg & : & \forall\iota_1.\forall\iota_2.\forall\iota_3.\ \iota_1 > 0 \supset (\underline{\mathbf{MUL}}(\iota_1,\iota_2,\iota_3) \rightarrow \underline{\mathbf{MUL}}(-\iota_1,\iota_2,-\iota_3))
\end{array}$$

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

$$0 * n = 0; (m+1) * n = m * n + n \ \text{ if } m >= 0; (-m) * n = -(m * n) \ \text{ if } m > 0.$$

In Figure 2, a function **concat** of the following type for concatenating a list of

```
// (...) is used to form tuples; e.g., () is the empty tuple
// | is used like a comma, which separates proofs from values
// {...} means universal quantification
// [...] means existential quantification
fun concat1 {a:type} {m, n:int | m >= 0; n >= 0}
  (xxs: list (list (a, n), m))
  : [p:int | p >=0 ] (MUL (m, n, p) | list (a, p)) =
  case+ xxs of
    | nil () => (MULbas | nil ())
    | cons (xs, xss) =>
      let val (pf | res) = concat1 xss in
         (MULind pf | xs @ res)
      end
```

Fig. 2. An implementation of the list concatenation function in ATS

lists is implemented:

$$\begin{array}{l}\forall\alpha.\forall\iota_1.\forall\iota_2.\ (\iota_1 \geq 0 \wedge \iota_2 \geq 0) \supset \\ \quad (\mathbf{list}(\mathbf{list}(\alpha, \iota_2), \iota_1) \rightarrow \exists\iota.\ \underline{\mathbf{MUL}}(\iota_1, \iota_2, \iota) * \mathbf{list}(\alpha, \iota))\end{array}$$

When applied to a list of length I_1 in which each element is a list of length I_2, this function returns a pair (pf, v), where v is a list of length I and pf (of the prop $\underline{\mathbf{MUL}}(I_1, I_2, I)$) is what we call a proof (in contrast to a program), which provides a witness to $I = I_1 * I_2$. Please find many more programming examples as such written in ATS [Xi], a language with a highly expressive type system rooted in the framework *Applied Type System* ($\mathcal{ATS}$) [Xi04]. Proofs can often be large and expensive to construct and should be erased before run-time. Note that we extract nothing from proofs. This style of programing, which we call *programming with theorem proving*, is rather different from the paradigm of program extraction (from proofs) as is supported in NuPrl [C$^+$86] or Coq [PM89,DFH$^+$93,BC04]). For instance, a function like **concat** can be effectful (though it is not) and need *not* to be terminating (though it is). To support the construction of programs involving effects (e.g., nontermination, exceptions and references, nondeterminism) is probably one of the most crucial issues in the design of ATS.

In order to guarantee that proofs in a program can be erased without altering the dynamic semantics of the program, a design is adopted in ATS that completely separates proofs from programs. Generally speaking, props (i.e., types for proofs) are introduced that can only be assigned to proofs, which are verified to be total (i.e., pure and terminating) in the type system of ATS, and programs, *even if they are total*, are disallowed in the construction of proofs. In short, programs may contain proofs but proofs cannot contain programs. While this design is conceptually simple, it leads to a rather duplicated presentation of various rules (e.g., typing rules and evaluation rules) for proofs and programs [CX05]. More seriously, it also complicates certain cases of proof construction that can be made significantly simpler if total programs are allowed to occur inside proofs. We will present an example in Section 4 to clarify this point.

In this paper, we follow the design of program extraction in Coq, where proofs can occur in programs and vice versa. The primary contribution of this paper lies in a novel design for unifying proofs and programs in an effectful programming language (in contrast to a theorem proving system like Coq). The developed formalism for supporting this design is already employed in ATS/Geizella, the current implementation of ATS [Xi]. However, for brevity, we can only present the essential idea behind this design in a simply typed setting and then outline an extension that accommodates dependent types as well as polymorphic types. We organize the rest of the paper as follows. In Section 2, we present a language $\mathcal{L}_0$ based on the simply typed lambda-calculus. A form of attribute types are supported in $\mathcal{L}_0$ to determine whether expressions assigned such types can be erased at compile-time without affecting the dynamic semantics of a program. We then outline an extension of $\mathcal{L}_0$ to $\mathcal{L}_0^{\forall,\exists}$ in Section 3 to support both dependent types and polymorphic types. In Section 4, we give a short but realistic example to illustrate a need for unifying proofs with programs. Lastly, we mention some closely related work and conclude.

erasure bits	$b ::= 0 \mid 1$
effect bits	$t ::= 0 \mid 1$
types	$\tau ::= \delta \mid \mathbf{1} \mid \tau_1 * \tau_2 \mid \tau_1 \to^t \tau_2$
a-types	$\underline{\hat{\tau}} ::= (\underline{\tau})^b$
a-type cores	$\underline{\tau} ::= \delta \mid \mathbf{1} \mid \underline{\hat{\tau}}_1 * \underline{\hat{\tau}}_2 \mid \underline{\hat{\tau}}_1 \to^t \underline{\hat{\tau}}_2$
expr.	$e ::= x \mid f \mid c(e) \mid \mathbf{if}(e_0, e_1, e_2) \mid \langle\rangle \mid \langle e_1, e_2\rangle \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid$ $\mathbf{lam}\, x.\, e \mid \mathbf{fix}\, f.\, e \mid \mathbf{app}(e_1, e_2)$
a-expr.	$\underline{\hat{e}} ::= (\underline{e})^b$
a-expr. cores	$\underline{e} ::= x \mid f \mid c(\underline{\hat{e}}) \mid \mathbf{if}(\underline{\hat{e}}_0, \underline{\hat{e}}_1, \underline{\hat{e}}_2) \mid \langle\rangle \mid \langle \underline{\hat{e}}_1, \underline{\hat{e}}_2\rangle \mid \mathbf{fst}(\underline{\hat{e}}) \mid \mathbf{snd}(\underline{\hat{e}}) \mid$ $\mathbf{lam}\, x.\, \underline{\hat{e}} \mid \mathbf{fix}\, f.\, \underline{\hat{e}} \mid \mathbf{app}(\underline{\hat{e}}_1, \underline{\hat{e}}_2)$
values	$v ::= x \mid cc(v) \mid \langle\rangle \mid \langle v_1, v_2\rangle \mid \mathbf{lam}\, x.\, e$
a-values	$\underline{\hat{v}} ::= (\underline{v})^b$
a-value cores	$\underline{v} ::= x \mid cc(\underline{\hat{v}}) \mid \langle\rangle \mid \langle \underline{\hat{v}}_1, \underline{\hat{v}}_2\rangle \mid \mathbf{lam}\, x.\, \underline{\hat{e}}$
contexts	$\Gamma ::= \emptyset \mid \Gamma, xf : \underline{\hat{\tau}}$
substitutions	$\theta ::= [] \mid \theta[x \mapsto \underline{\hat{v}}] \mid \theta[f \mapsto \underline{\hat{e}}]$

Fig. 3. The syntax for $\mathcal{L}_0$

2 Formal Development

We present a language $\mathcal{L}_0$ based on the simply typed lambda-calculus to formally introduce a form of attributive types. The syntax for $\mathcal{L}_0$ is given in Figure 3. We use both b (erasure bits) and t (effect bits) to range over 0 and 1. Given two (erasure) bits b_1 and b_2, $b_1 \otimes b_2$ is a bit, which equals 1 if and only if $b_1 = b_2 = 1$. Given two (effect) bits t_1 and t_2, $t_1 \oplus t_2$ is a bit, which equals 0 if and only if $t_1 = t_2 = 0$. So, $\otimes$ and $\oplus$ correspond to boolean product and sum, respectively.

We use δ, τ, $\underline{\tau}$ and $\hat{\underline{\tau}}$ for base types (e.g., **bool** for booleans and **int** for integers), types, a-type (i.e., attributive type) cores and a-types, respectively. Similarly, we use e, $\underline{e}$ and $\hat{\underline{e}}$ for expressions, a-expression cores, a-expressions, respectively. Given $\hat{\underline{\tau}} = (\underline{\tau})^b$, we write $bit(\hat{\underline{\tau}})$ for b, $core(\hat{\underline{\tau}})$ for $\underline{\tau}$, and $(\hat{\underline{\tau}})^{b_0}$ for $(\underline{\tau})^{b\otimes b_0}$. Similarly, given $\hat{\underline{e}} = (\underline{e})^b$, we write $bit(\hat{\underline{e}})$ for b, $core(\hat{\underline{e}})$ for $\underline{e}$, and $(\hat{\underline{e}})^{b_0}$ for $(\underline{e})^{b\otimes b_0}$. So both $(\hat{\underline{\tau}})^b$ and $(\hat{\underline{e}})^b$ are just syntactic sugar. It will soon become clear that an a-type $\hat{\underline{\tau}}$ can only be assigned to an a-expression $\hat{\underline{e}}$ such that $bit(\hat{\underline{\tau}}) = bit(\hat{\underline{e}})$, and $bit(\hat{\underline{e}}) = 0$ means that $\hat{\underline{e}}$ can be erased from any program containing $\hat{\underline{e}}$ without altering the dynamic semantics of the program. We say that $\hat{\underline{e}}$ is erasable if $bit(\hat{\underline{e}}) = 0$.

Clearly, an a-expression cannot be erased if its evaluation may generate effects at run-time. To address this issue, we design a type system based on the notion of types with effects [JG91] to track whether evaluating an a-expression may generate effects. Given an a-type core $\underline{\tau} = \hat{\underline{\tau}}_1 \rightarrow^t \hat{\underline{\tau}}_2$, a call to a function of a-type $(\underline{\tau})^b$ may generate effects *only if* $t = 1$. Note that nonterminating evaluation is the sole kind of effect in $\mathcal{L}_0$, but more can be easily added when $\mathcal{L}_0$ is extended. For instance, in ATS, we also track effects caused by raising exceptions, accessing references, aborting program execution (abnormally), etc.

We use x for a **lam**-variable and f for a **fix**-variable, and xf for either an x or f. We use c for a constant, which is either a constant constructor cc or a constant function cf. The a-expressions in Figure 3 are standard except for the erasure bits they carry.

We now assign a dynamic semantics to $\mathcal{L}_0$. The evaluation contexts in $\mathcal{L}_0$ are defined as follows.

$$\begin{aligned}\text{eval. ctx. } E ::= {} & [] \mid (E)^b \mid (c(E))^b \mid (\mathbf{if}(E, \hat{\underline{e}}_1, \hat{\underline{e}}_2))^b \mid (\langle E, \hat{\underline{e}}\rangle)^b \mid (\langle \hat{\underline{v}}, E\rangle)^b \mid \\ & (\mathbf{fst}(E))^b \mid (\mathbf{snd}(E))^b \mid (\mathbf{app}(E, \hat{\underline{e}}))^b \mid (\mathbf{app}(\hat{\underline{v}}, E))^b\end{aligned}$$

The redexes in $\mathcal{L}_0$ and their reducts are defined below.

Definition 1. *(Redexes) We define redexes and their reducts as follows.*

- $((\underline{v})^{b_1})^{b_2}$ *is a redex, and its reduct is* $(\underline{v})^{b_1\otimes b_2}$.
- $(\mathbf{if}((true)^{b_0}, \hat{\underline{e}}_1, \hat{\underline{e}}_2))^b$ *is redex, and its reduct is* $(\hat{\underline{e}}_1)^b$.
- $(\mathbf{if}((false)^{b_0}, \hat{\underline{e}}_1, \hat{\underline{e}}_2))^b$ *is redex, and its reduct is* $(\hat{\underline{e}}_2)^b$.
- $(\mathbf{app}((\mathbf{lam}\, x.\, \hat{\underline{e}})^{b_0}, \hat{\underline{v}}))^b$ *is a redex, and its reduct is* $(\hat{\underline{e}}[x \mapsto \hat{\underline{v}}])^b$.
- $(\mathbf{fst}((\langle \hat{\underline{v}}_1, \hat{\underline{v}}_2\rangle)^{b_0}))^b$ *is a redex, and its reduct is* $(\hat{\underline{v}}_1)^b$.
- $(\mathbf{snd}((\langle \hat{\underline{v}}_1, \hat{\underline{v}}_2\rangle)^{b_0}))^b$ *is a redex, and its reduct is* $(\hat{\underline{v}}_2)^b$.
- $(\mathbf{fix}\, f.\, \hat{\underline{e}})^b$ *is a redex and its reduct is* $\hat{\underline{e}}[f \mapsto (\mathbf{fix}\, f.\, \hat{\underline{e}})^b]$.

Given $\hat{\underline{e}}_1$ and $\hat{\underline{e}}_2$, we write $\hat{\underline{e}}_1 \hookrightarrow \hat{\underline{e}}_2$ to mean that $\hat{\underline{e}}_1$ reduces to $\hat{\underline{e}}_2$, that is, $\hat{\underline{e}}_1 = E[\hat{\underline{e}}]$ and $\hat{\underline{e}}_2 = E[\hat{\underline{e}}']$ for some redex $\hat{\underline{e}}$ and its reduct $\hat{\underline{e}}'$. We may also use $\hookrightarrow$ for the single step call-by-value evaluation on expressions, which is completely standard. As usual, we use $\hookrightarrow^+$ for the transitive closure of $\hookrightarrow$, and $\hookrightarrow^*$ for the transitive and reflexive closure of $\hookrightarrow$.

We use $\Gamma \vdash^t \hat{\underline{e}} : \hat{\underline{\tau}}$ for a typing judgment in $\mathcal{L}_0$ that assigns the a-type $\hat{\underline{\tau}}$ to the a-expression $\hat{\underline{e}}$, where the bit t is used to indicate whether evaluating $\hat{\underline{e}}$ may generate any effects. The static semantics for $\mathcal{L}_0$ is given by the typing

$$\frac{\Gamma(xf) = \underline{\hat{\tau}} \quad b = bit(\underline{\hat{\tau}})}{\Gamma \vdash^0 (x)^b : \underline{\hat{\tau}}} \ \textbf{(ty-lam-var)} \qquad \frac{\Gamma(xf) = \underline{\hat{\tau}} \quad bit(\underline{\hat{\tau}}) = 1}{\Gamma \vdash^1 (f)^1 : \underline{\hat{\tau}}} \ \textbf{(ty-fix-var)}$$

$$\frac{\vdash c : \underline{\hat{\tau}}_1 \xrightarrow{t_0} \underline{\hat{\tau}}_2 \quad \Gamma \vdash^t \underline{\hat{e}} : \underline{\hat{\tau}}_1 \quad t_0 \oplus t \leq b}{\Gamma \vdash^{t_0 \oplus t} (c(\underline{\hat{e}}))^b : \underline{\hat{\tau}}_2} \ \textbf{(ty-const)}$$

$$\frac{\Gamma \vdash^{t_0} \underline{\hat{e}}_0 : (\mathbf{bool})^{b_0} \quad \Gamma \vdash^{t_1} \underline{\hat{e}}_1 : \underline{\hat{\tau}} \quad \Gamma \vdash^{t_2} \underline{\hat{e}}_2 : \underline{\hat{\tau}} \quad t_0 \oplus t_1 \oplus t_2 \leq b = bit(\underline{\hat{\tau}}) \otimes b_0}{\Gamma \vdash^{t_0 \oplus t_1 \oplus t_2} (\mathbf{if}(\underline{\hat{e}}_0, \underline{\hat{e}}_1, \underline{\hat{e}}_2))^b : (\underline{\hat{\tau}})^{b_0}} \ \textbf{(ty-if)}$$

$$\frac{\Gamma \vdash^{t_1} \underline{\hat{e}}_1 : \underline{\hat{\tau}}_1 \quad \Gamma \vdash^{t_2} \underline{\hat{e}}_2 : \underline{\hat{\tau}}_2 \quad t_1 \oplus t_2 \leq b}{\Gamma \vdash^{t_1 \oplus t_2} (\langle \underline{\hat{e}}_1, \underline{\hat{e}}_2 \rangle)^b : (\langle \underline{\hat{\tau}}_1, \underline{\hat{\tau}}_2 \rangle)^b} \ \textbf{(ty-tup)}$$

$$\frac{\Gamma \vdash^t \underline{\hat{e}} : (\underline{\hat{\tau}}_1 * \underline{\hat{\tau}}_2)^{b_0} \quad t \leq b = bit(\underline{\hat{\tau}}_1) \otimes b_0}{\Gamma \vdash^t (\mathbf{fst}(\underline{\hat{e}}))^b : (\underline{\hat{\tau}}_1)^{b_0}} \ \textbf{(ty-fst)}$$

$$\frac{\Gamma \vdash^t \underline{\hat{e}} : (\underline{\hat{\tau}}_1 * \underline{\hat{\tau}}_2)^{b_0} \quad t \leq b = bit(\underline{\hat{\tau}}_2) \otimes b_0}{\Gamma \vdash^t (\mathbf{snd}(\underline{\hat{e}}))^b : (\underline{\hat{\tau}}_2)^{b_0}} \ \textbf{(ty-snd)}$$

$$\frac{\Gamma, x : \underline{\hat{\tau}}_1 \vdash^t \underline{\hat{e}} : \underline{\hat{\tau}}_2}{\Gamma \vdash^0 (\mathbf{lam}\, x.\, \underline{\hat{e}})^b : (\underline{\hat{\tau}}_1 \xrightarrow{t} \underline{\hat{\tau}}_2)^b} \ \textbf{(ty-lam)} \qquad \frac{\Gamma, f : \underline{\hat{\tau}} \vdash^t \underline{\hat{e}} : \underline{\hat{\tau}} \quad t \leq b = bit(\underline{\hat{\tau}})}{\Gamma \vdash^t (\mathbf{fix}\, f.\, \underline{\hat{e}})^b : \underline{\hat{\tau}}} \ \textbf{(ty-fix)}$$

$$\frac{\Gamma \vdash^{t_1} \underline{\hat{e}}_1 : (\underline{\hat{\tau}}_1 \xrightarrow{t} \underline{\hat{\tau}}_2)^{b_0} \quad \Gamma \vdash^{t_2} \underline{\hat{e}}_2 : \underline{\hat{\tau}}_1 \quad t_1 \oplus t_2 \oplus t \leq b = bit(\underline{\hat{\tau}}_2) \otimes b_0}{\Gamma \vdash^{t_1 \oplus t_2 \oplus t} (\mathbf{app}(\underline{\hat{e}}_1, \underline{\hat{e}}_2))^b : (\underline{\hat{\tau}}_2)^{b_0}} \ \textbf{(ty-app)}$$

$$\frac{\Gamma \vdash^0 \underline{\hat{e}} : \underline{\hat{\tau}}}{\Gamma \vdash^0 (\underline{\hat{e}})^0 : (\underline{\hat{\tau}})^0} \ \textbf{(ty-erase)}$$

Fig. 4. The typing rules for $\mathcal{L}_0$

rules in Figure 4, for some of which we provide brief explanation as follows. The rule **(ty-fix-var)** indicates that a fix-variable is considered effectful; in the rule **(ty-const)**, we write $\vdash c : \underline{\hat{\tau}}_1 \xrightarrow{t_0} \underline{\hat{\tau}}_2$ to mean that c is assigned the a-type core $\underline{\hat{\tau}}_1 \xrightarrow{t_0} \underline{\hat{\tau}}_2$, for instance, by some kind of signature; the rule **(ty-erase)** essentially means that an a-expression $\underline{\hat{e}}$ can be erased if the evaluation of $\underline{\hat{e}}$ is guaranteed to be free of effects; from the rule **(ty-if)**, it is clear that an if-expression must be erasable if its condition is erasable; the rule **(ty-app)** indicates that an application is erasable if the function in the application is.

Proposition 1. *If $\Gamma \vdash^t \underline{\hat{v}} : \underline{\hat{\tau}}$ is derivable, then $t = 0$.*

Proof. By an inspection of the typing rules in Figure 4.

Proposition 2. *If $\Gamma \vdash^t \underline{\hat{e}} : \underline{\hat{\tau}}$ is derivable, then $t \leq bit(\underline{\hat{e}}) = bit(\underline{\hat{\tau}})$.*

Proof. Note that $bit((\underline{\hat{\tau}})^b) = bit(\underline{\hat{\tau}}) \otimes b$ holds for any a-type $\underline{\hat{\tau}}$ and bit b. This proposition follows from an inspection of the rules in Figure 4.

Lemma 1. *(Canonical Forms) Assume that $\emptyset \vdash^0 (\underline{v})^b : (\underline{\tau})^b$ is derivable.*

1. *If $\underline{\tau} = \delta$ for some base type δ, then $\underline{v}$ is of the form $cc(\underline{\hat{v}}_0)$ for some constructor cc associated with δ, that is, cc is given an a-type core of the form $\underline{\hat{\tau}}_0 \to \delta$ for some $\underline{\hat{\tau}}_0$.*
2. *If $\underline{\tau} = \underline{\hat{\tau}}_1 * \underline{\hat{\tau}}_2$, then $\underline{v}$ is of the form $\langle \underline{\hat{v}}_1, \underline{\hat{v}}_2 \rangle$.*
3. *If $\underline{\tau} = \underline{\hat{\tau}}_1 \to^t \underline{\hat{\tau}}_2$, then $\underline{v}$ is of the form $\mathbf{lam}\, x.\, \underline{\hat{e}}$.*

Proof. By an inspection of the typing rules in Figure 4.

Lemma 2. *(Substitution) We have the following.*

1. *Assume that both $\Gamma \vdash^0 \underline{\hat{v}} : \underline{\hat{\tau}}_1$ and $\Gamma, x : \underline{\hat{\tau}}_1 \vdash^t \underline{\hat{e}} : \underline{\hat{\tau}}_2$ are derivable. Then $\Gamma \vdash^t \underline{\hat{e}}[x \mapsto \underline{\hat{v}}] : \underline{\hat{\tau}}_2$ is also derivable.*
2. *Assume that both $\Gamma \vdash^{t_1} \underline{\hat{e}}_1 : \underline{\hat{\tau}}_1$ and $\Gamma, f : \underline{\hat{\tau}}_1 \vdash^{t_2} \underline{\hat{e}}_2 : \underline{\hat{\tau}}_2$ are derivable. Then $\Gamma \vdash^{t'_2} \underline{\hat{e}}_2[f \mapsto \underline{\hat{e}}_1] : \underline{\hat{\tau}}_2$ is derivable for some $t'_2 \leq t_2$.*

Proof. By structural induction on the derivations of $\Gamma, x : \underline{\hat{\tau}}_1 \vdash^t \underline{\hat{e}} : \underline{\hat{\tau}}_2$ and $\Gamma, f : \underline{\hat{\tau}}_1 \vdash^{t_2} \underline{\hat{e}}_2 : \underline{\hat{\tau}}_2$, respectively.

Theorem 1. *(Subject Reduction) Assume that $\emptyset \vdash^{t_1} \underline{\hat{e}}_1 : \underline{\hat{\tau}}$ is derivable and $\underline{\hat{e}}_1 \hookrightarrow \underline{\hat{e}}_2$ holds. Then $\emptyset \vdash^{t_2} \underline{\hat{e}}_2 : \underline{\hat{\tau}}$ is derivable for some $t_2 \leq t_1$.*

Proof. By structural induction on the derivation $\mathcal{D}$ of $\emptyset \vdash^{t_1} \underline{\hat{e}}_1 : \underline{\hat{\tau}}$. Lemma 2 is needed when handling the case where the last applied rule in $\mathcal{D}$ is **(ty-app)** or **(ty-fix)**.

Theorem 2. *(Progress) Assume that $\emptyset \vdash^t \underline{\hat{e}}_1 : \underline{\hat{\tau}}$ is derivable. Then either $\underline{\hat{e}}_1$ is an a-value or $\underline{\hat{e}}_1 \hookrightarrow \underline{\hat{e}}_2$ for some a-expression $\underline{\hat{e}}_2$.*

Proof. The theorem follows from structural induction on the derivation $\mathcal{D}$ of $\emptyset \vdash^t \underline{\hat{e}}_1 : \underline{\hat{\tau}}$.

With Theorem 1 and Theorem 2, it is clear that for each well-typed a-expression $\underline{\hat{e}}$, that is, $\emptyset \vdash \underline{\hat{e}} : \underline{\hat{\tau}}$ is derivable for some $\underline{\hat{\tau}}$, the evaluation of $\underline{\hat{e}}$ either leads to an a-value or it continues forever. So the type soundness of $\mathcal{L}_0$ is established. Of course, we can also prove the type soundness of $\mathcal{L}_0$ by simply ignoring erasure bits. However, we need Theorem 1 and Theorem 2 to prove Theorem 5, a main result in the paper that justifies proof erasure.

Definition 2. *(Reducibility) Given an a-expression $\underline{\hat{e}}$ and an a-type $\underline{\hat{\tau}} = (\underline{\tau})^{b_0}$, we say that $\underline{\hat{e}}$ is reducible of $\underline{\hat{\tau}}$ if $\underline{\hat{e}}\!\downarrow$, that is, there is no infinite reduction sequence from $\underline{\hat{e}}$, and*

1. *$\underline{\tau} = \delta$ for some base type δ; or*
2. *$\underline{\tau} = \underline{\hat{\tau}}_1 * \underline{\hat{\tau}}_2$ for some $\underline{\hat{\tau}}_1$ and $\underline{\hat{\tau}}_2$ and for any $\underline{\hat{v}}_1$ and $\underline{\hat{v}}_2$, $\underline{\hat{e}} \hookrightarrow^* (\langle \underline{\hat{v}}_1, \underline{\hat{v}}_2 \rangle)^b$ implies that $\underline{\hat{v}}_1$ and $\underline{\hat{v}}_2$ are reducible of $\underline{\hat{\tau}}_1$ and $\underline{\hat{\tau}}_2$, respectively; or*
3. *$\underline{\tau} = \underline{\hat{\tau}}_1 \to^0 \underline{\hat{\tau}}_2$ for some $\underline{\hat{\tau}}_1$ and $\underline{\hat{\tau}}_2$ and for any $\underline{\hat{e}}_0$, $\underline{\hat{e}} \hookrightarrow^* (\mathbf{lam}\, x.\, \underline{\hat{e}}_0)^b$ implies that $\underline{\hat{e}}_0[x \mapsto \underline{\hat{v}}]$ is reducible of $\underline{\hat{\tau}}_2$ for every $\underline{\hat{v}}$ reducible of a-type $\underline{\hat{\tau}}_1$; or*
4. *$\underline{\tau} = \underline{\hat{\tau}}_1 \to^1 \underline{\hat{\tau}}_2$ for some $\underline{\hat{\tau}}_1$ and $\underline{\hat{\tau}}_2$.*

Given a substitution θ and a context Γ of the same domain, we say that θ is reducible of Γ if $\theta(xf)$ is reducible of $\Gamma(xf)$ for every $xf \in \mathbf{dom}(\theta) = \mathbf{dom}(\Gamma)$.

It should be stressed that $\underline{\hat{e}}$ being reducible of $\underline{\hat{\tau}}$ does *not* imply that $\underline{\hat{\tau}}$ can be assigned to $\underline{\hat{e}}$. For instance, according to the definition, every value (including function value) is reducible of $(\delta)^{b_0}$ for every base type δ. Also, it is clear that erasure bits play no role in the definition of reducibility. More precisely, if $\underline{\hat{e}}$ is reducible of $(\underline{\tau})^b$ for some b, then it is reducible of $(\underline{\tau})^b$ for any b.

Proposition 3. *We have the following.*

1. *If $\underline{\hat{e}}_1$ is reducible of $\underline{\hat{\tau}}$ and $\underline{\hat{e}}_1 \hookrightarrow \underline{\hat{e}}_2$, then $\underline{\hat{e}}_2$ is also reducible of $\underline{\hat{\tau}}$.*
2. *If $\underline{\hat{e}}$ is reducible of $\underline{\hat{\tau}}$ whenever $\underline{\hat{e}}_1 \hookrightarrow \underline{\hat{e}}$ holds, then $\underline{\hat{e}}_1$ is reducible of $\underline{\hat{\tau}}$.*
3. *If $\underline{\hat{e}}$ is reducible of $\underline{\hat{\tau}}$, then $(\underline{\hat{e}})^b$ is reducible of $(\underline{\hat{\tau}})^b$.*
4. *If $\underline{\hat{e}}_1$ and $\underline{\hat{e}}_2$ are reducible of $\underline{\hat{\tau}}_1$ and $\underline{\hat{\tau}}_2$, respectively, then $(\langle\underline{\hat{e}}_1, \underline{\hat{e}}_2\rangle)^b$ is reducible of $(\underline{\hat{\tau}}_1 * \underline{\hat{\tau}}_2)^b$.*
5. *If $\underline{\hat{e}}$ is reducible of $(\underline{\hat{\tau}}_1 * \underline{\hat{\tau}}_2)^{b_0}$, then for $b = bit(\underline{\hat{\tau}}_1) \otimes b_0$, $(\mathbf{fst}(\underline{\hat{e}}))^b$ is reducible of $(\underline{\hat{\tau}}_1)^{b_0}$.*
6. *If $\underline{\hat{e}}$ is reducible of $(\underline{\hat{\tau}}_1 * \underline{\hat{\tau}}_2)^{b_0}$, then for $b = bit(\underline{\hat{\tau}}_2) \otimes b_0$, $(\mathbf{snd}(\underline{\hat{e}}))^b$ is reducible of $(\underline{\hat{\tau}}_2)^{b_0}$.*
7. *If $\underline{\hat{e}}_1$ and $\underline{\hat{e}}_2$ are reducible of $(\underline{\hat{\tau}}_1 \to^0 \underline{\hat{\tau}}_2)^{b_0}$ and $\underline{\hat{\tau}}_1$, respectively, then for $b = bit(\underline{\hat{\tau}}_2) \otimes b_0$, $(\mathbf{app}(\underline{\hat{e}}_1, \underline{\hat{e}}_2))^b$ is reducible of $(\underline{\hat{\tau}}_2)^{b_0}$.*

Proof. Both (1) and (2) follow from the definition of reducibility immediately. As for (3), it follows by structural induction on $\underline{\hat{\tau}}$.

We now prove (4). Clearly, both $\underline{\hat{e}}_1\downarrow$ and $\underline{\hat{e}}_2\downarrow$ hold. So $(\langle\underline{\hat{e}}_1, \underline{\hat{e}}_2\rangle)^b\downarrow$ holds as well. Suppose $(\langle\underline{\hat{e}}_1, \underline{\hat{e}}_2\rangle)^b \hookrightarrow^* (\langle\underline{\hat{v}}_1, \underline{\hat{v}}_2\rangle)^b$. Then $\underline{\hat{e}}_1 \hookrightarrow^* \underline{\hat{v}}_1$ and $\underline{\hat{e}}_2 \hookrightarrow^* \underline{\hat{v}}_2$. By (1), $\underline{\hat{v}}_1$ and $\underline{\hat{v}}_2$ are reducible of $\underline{\hat{\tau}}_1$ and $\underline{\hat{\tau}}_2$, respectively. By definition, $\underline{\hat{e}}$ is reducible of $(\underline{\hat{\tau}}_1 * \underline{\hat{\tau}}_2)^b$.

We leave out the routine proofs for (5), (6) and (7).

Lemma 3. *Assume that $\Gamma \vdash^0 \underline{\hat{e}} : \underline{\hat{\tau}}$ is derivable and θ is reducible of Γ. Then $\underline{\hat{e}}[\theta]$ is reducible of $\underline{\hat{\tau}}$.*

Proof. We proceed by structural induction on the derivation $\mathcal{D}$ of $\Gamma \vdash^0 \underline{\hat{e}} : \underline{\hat{\tau}}$.

- Assume that $\mathcal{D}$ is of the following form:

$$\frac{\mathcal{D}_1 :: \Gamma \vdash^0 \underline{\hat{e}}_1 : \underline{\hat{\tau}}_1 \quad \mathcal{D}_2 :: \Gamma \vdash^0 \underline{\hat{e}}_2 : \underline{\hat{\tau}}_2 \quad 0 \oplus 0 \leq b}{\Gamma \vdash^0 (\langle\underline{\hat{e}}_1, \underline{\hat{e}}_2\rangle)^b : (\langle\underline{\hat{\tau}}_1, \underline{\hat{\tau}}_2\rangle)^b} \ \textbf{(ty-tup)}$$

  where $\underline{\hat{e}} = (\langle\underline{\hat{e}}_1, \underline{\hat{e}}_2\rangle)^b$. By induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}_2$, we know that $\underline{\hat{e}}_1[\theta]$ and $\underline{\hat{e}}_2[\theta]$ are reducible of $\underline{\hat{\tau}}_1$ and $\underline{\hat{\tau}}_2$, respectively. Hence, $\underline{\hat{e}}[\theta] = (\langle\underline{\hat{e}}_1[\theta], \underline{\hat{e}}_2[\theta]\rangle)^b$ is reducible of $(\underline{\hat{\tau}}_1 * \underline{\hat{\tau}}_2)^b$ by Proposition 3 (4).
- Assume that $\mathcal{D}$ is of the following form:

$$\frac{\Gamma \vdash^0 \underline{\hat{e}}_1 : (\underline{\hat{\tau}}_1 \to^t \underline{\hat{\tau}}_2)^{b_0} \quad \Gamma \vdash^0 \underline{\hat{e}}_2 : \underline{\hat{\tau}}_1 \quad 0 \oplus 0 \oplus 0 \leq b = bit(\underline{\hat{\tau}}_2) \otimes b_0}{\Gamma \vdash^0 (\mathbf{app}(\underline{\hat{e}}_1, \underline{\hat{e}}_2))^b : (\underline{\hat{\tau}}_2)^{b_0}} \ \textbf{(ty-app)}$$

$$
\begin{array}{c}
|(\underline{e})^0| = \langle\rangle \qquad |(\underline{e})^1| = |\underline{e}| \qquad |xf| = xf \qquad |c(\underline{\hat{e}})| = c(|\underline{\hat{e}}|) \\
|\mathbf{if}(\underline{\hat{e}}_0, \underline{\hat{e}}_1, \underline{\hat{e}}_2)| = \mathbf{if}(|\underline{\hat{e}}_0|, |\underline{\hat{e}}_1|, |\underline{\hat{e}}_2|) \qquad |\langle\rangle| = \langle\rangle \\
|\langle \underline{\hat{e}}_1, \underline{\hat{e}}_2 \rangle| = \langle |\underline{\hat{e}}_1|, |\underline{\hat{e}}_2| \rangle \qquad |\mathbf{fst}(\underline{\hat{e}})| = \mathbf{fst}(|\underline{\hat{e}}|) \qquad |\mathbf{snd}(\underline{\hat{e}})| = \mathbf{snd}(|\underline{\hat{e}}|) \\
|\mathbf{lam}\, x.\, \underline{\hat{e}}| = \mathbf{lam}\, x.\, |\underline{\hat{e}}| \qquad |\mathbf{fix}\, f.\, \underline{\hat{e}}| = \mathbf{fix}\, f.\, |\underline{\hat{e}}| \qquad |\mathbf{app}(\underline{\hat{e}}_1, \underline{\hat{e}}_2)| = \mathbf{app}(|\underline{\hat{e}}_1|, |\underline{\hat{e}}_2|)
\end{array}
$$

Fig. 5. The erasure function on a-expression cores and a-expressions

where $\underline{\hat{e}} = (\mathbf{app}(\underline{\hat{e}}_1, \underline{\hat{e}}_2))^b$ and $\underline{\hat{\tau}} = (\underline{\hat{\tau}}_2)^{b_0}$. Then by induction hypothesis, $\underline{\hat{e}}_1[\theta]$ and $\underline{\hat{e}}_2[\theta]$ are reducible of $(\underline{\hat{\tau}}_1 \xrightarrow{t} \underline{\hat{\tau}}_2)^{b_0}$ and $\underline{\hat{\tau}}_2$, respectively. By Proposition 3 (7), $\underline{\hat{e}}[\theta]$ is reducible of $\underline{\hat{\tau}}$.

The rest of cases can be handled similarly (by applying Proposition 3).

Theorem 3. *(Totality) Assume that $\emptyset \vdash^0 \underline{\hat{e}} : \underline{\hat{\tau}}$ is derivable. Then $\underline{\hat{e}}\downarrow$ holds.*

Proof. By Lemma 3, $\underline{\hat{e}}$ is reducible of $\underline{\hat{\tau}}$. By the definition of reducibility, $\underline{\hat{e}}\downarrow$ holds.

When $\mathcal{L}_0$ is extended with dependent types, it becomes a great deal more involved to prove a corresponding version of Theorem 3. The technique for doing so is developed in [Xi02].

A function $|\cdot|$ is defined in Figure 5 that erases a-expressions into expressions. Note that the erasure of an a-expression $\underline{\hat{e}}$ is $\langle\rangle$ if $bit(\underline{\hat{e}}) = 0$. As is desired, erasure commutes with substitution.

Proposition 4. *We have $|\underline{\hat{e}}_2[xf \mapsto \underline{\hat{e}}_1]| = |\underline{\hat{e}}_2|[xf \mapsto |\underline{\hat{e}}_1|]$ for any a-expressions $\underline{\hat{e}}_1$ and $\underline{\hat{e}}_2$.*

Proof. This follows from the definition of the erasure function $|\cdot|$ immediately.

The soundness of erasure is stated and proven as follows.

Theorem 4. *(Soundness of Erasure) Assume that $\emptyset \vdash^t \underline{\hat{e}} : \underline{\hat{\tau}}$ is derivable. If $\underline{\hat{e}} \hookrightarrow \underline{\hat{e}}'$, then $|\underline{\hat{e}}| \hookrightarrow^{0/1} |\underline{\hat{e}}'|$, that is, $|\underline{\hat{e}}| = |\underline{\hat{e}}'|$ or $|\underline{\hat{e}}| \hookrightarrow |\underline{\hat{e}}'|$.*

Proof. We proceed by structural induction on the derivation $\mathcal{D}$ of $\emptyset \vdash^t \underline{\hat{e}}_1 : \underline{\hat{\tau}}$.

– The derivation $\mathcal{D}$ is of the following form:

$$
\frac{\emptyset \vdash^{t_0} \underline{\hat{e}}_0 : (\mathbf{bool})^{b_0} \quad \emptyset \vdash^{t_1} \underline{\hat{e}}_1 : \underline{\hat{\tau}} \quad \emptyset \vdash^{t_2} \underline{\hat{e}}_2 : \underline{\hat{\tau}} \quad t_0 \oplus t_1 \oplus t_2 \leq b = bit(\underline{\hat{\tau}}) \otimes b_0}{\emptyset \vdash^{t_0 \oplus t_1 \oplus t_2} (\mathbf{if}(\underline{\hat{e}}_0, \underline{\hat{e}}_1, \underline{\hat{e}}_2))^b : (\underline{\hat{\tau}})^{b_0}} \ \textbf{(ty-if)}
$$

where $\underline{\hat{e}} = (\mathbf{if}(\underline{\hat{e}}_0, \underline{\hat{e}}_1, \underline{\hat{e}}_2))^b$. We have three subcases.

- $\underline{\hat{e}}_0 \hookrightarrow \underline{\hat{e}}_0'$ for some $\underline{\hat{e}}_0'$ and $\underline{\hat{e}}' = (\mathbf{if}(\underline{\hat{e}}_0', \underline{\hat{e}}_1, \underline{\hat{e}}_2))^b$. If $b = 0$, we are done since $|\underline{\hat{e}}| = |\underline{\hat{e}}'| = \langle\rangle$. If $b = 1$, then $|\underline{\hat{e}}| \hookrightarrow^{0/1} |\underline{\hat{e}}'|$ holds since we have $|\underline{\hat{e}}_0| \hookrightarrow^{0/1} |\underline{\hat{e}}_0'|$ by induction hypothesis.

- $\underline{\hat{e}}_0 = (\mathit{true})^{b_0}$. Then $\underline{\hat{e}}' = (\underline{\hat{e}}_1)^b$. If $b = 0$, then we are done since $|\underline{\hat{e}}| = |\underline{\hat{e}}'| = \langle\rangle$. If $b = 1$, then b_0 must equal 1 and thus $|\underline{\hat{e}}| = \mathbf{if}(\mathit{true}, |\underline{\hat{e}}_1|, |\underline{\hat{e}}_2|)$, which implies $|\underline{\hat{e}}| \hookrightarrow |\underline{\hat{e}}_1| = |\underline{\hat{e}}'|$.
- $\underline{\hat{e}}_0 = (\mathit{false})^{b_0}$. This case is similar to the previous one.

The rest of the cases can be handled similarly.

The following theorem implies that if the erasure of a well-typed a-expression $\underline{\hat{e}}$ in $\mathcal{L}_0$ evaluates to a value v, then $\underline{\hat{e}}$ evaluates to an a-value whose erasure is v.

Theorem 5. *(Completeness of Erasure) Assume that $\emptyset \vdash^t \underline{\hat{e}} : \underline{\hat{\tau}}$ is derivable.*

1. *If $|\underline{\hat{e}}|$ is a value, then $\underline{\hat{e}} \hookrightarrow^* \underline{\hat{v}}$ for some a-value such that $|\underline{\hat{v}}| = |\underline{\hat{e}}|$.*
2. *If $|\underline{\hat{e}}| \hookrightarrow e'$, then $\underline{\hat{e}} \hookrightarrow^+ \underline{\hat{e}}'$ for some $\underline{\hat{e}}'$ such that $|\underline{\hat{e}}'| = e'$.*

Proof. We first prove (1) by analyzing the structure of $\underline{\hat{e}}$.

- $\mathit{bit}(\underline{\hat{e}}) = 0$. Then $|\underline{\hat{e}}| = \langle\rangle$. By Proposition 2, $t = 0$, and by Theorem 3, $\underline{\hat{e}}\downarrow$ holds. So by Theorem 1 and Theorem 2, we have $\underline{\hat{e}} \hookrightarrow^* \underline{\hat{v}}$ for some a-value $\underline{\hat{v}}$ of a-type $\underline{\hat{\tau}}$. Clearly, $|\underline{\hat{v}}| = \langle\rangle$ as $\mathit{bit}(\underline{\hat{v}}) \leq \mathit{bit}(\underline{\hat{e}}) = 0$.
- $\mathit{bit}(\underline{\hat{e}}) = 1$. We present an interesting case where $\underline{\hat{e}} = (\langle \underline{\hat{e}}_1, \underline{\hat{e}}_2 \rangle)^1$. Since $|\underline{\hat{e}}| = \langle |\underline{\hat{e}}_1|, |\underline{\hat{e}}_2| \rangle$ is a value, both $|\underline{\hat{e}}_1|$ and $|\underline{\hat{e}}_2|$ are values. By induction hypothesis, $\underline{\hat{e}}_1 \hookrightarrow^* \underline{\hat{v}}_1$ and $\underline{\hat{e}}_2 \hookrightarrow^* \underline{\hat{v}}_2$ for some a-values $\underline{\hat{v}}_1$ and $\underline{\hat{v}}_2$ such that $|\underline{\hat{v}}_1| = |\underline{\hat{e}}_1|$ and $|\underline{\hat{v}}_2| = |\underline{\hat{e}}_2|$. Let $\underline{\hat{v}} = \langle \underline{\hat{v}}_1, \underline{\hat{v}}_2 \rangle$, and we have $\underline{\hat{e}} \hookrightarrow^* \underline{\hat{v}}$ and $|\underline{\hat{v}}| = |\underline{\hat{e}}|$. All other cases can be handled similarly.

We now prove (2) by structural induction on the derivation $\mathcal{D}$ of $\emptyset \vdash^t \underline{\hat{e}}_1 : \underline{\hat{\tau}}$.

- The derivation $\mathcal{D}$ is of the following form:

$$\frac{\emptyset \vdash^{t_1} \underline{\hat{e}}_1 : \underline{\hat{\tau}}_1 \quad \emptyset \vdash^{t_2} \underline{\hat{e}}_2 : \underline{\hat{\tau}}_2 \quad t_1 \oplus t_2 \leq b}{\emptyset \vdash^{t_1 \oplus t_2} (\langle \underline{\hat{e}}_1, \underline{\hat{e}}_2 \rangle)^b : (\langle \underline{\hat{\tau}}_1, \underline{\hat{\tau}}_2 \rangle)^b} \ \textbf{(ty-tup)}$$

  where $\underline{\hat{e}} = (\langle \underline{\hat{e}}_1, \underline{\hat{e}}_2 \rangle)^b$. Clearly, $b = 1$ since $|\underline{\hat{e}}|$ cannot be $\langle\rangle$. There are two subcases.
  - $|\underline{\hat{e}}_1|$ is a value. Then by (1), there exists $\underline{\hat{v}}_1$ such that $\underline{\hat{e}}_1 \hookrightarrow^* \underline{\hat{v}}_1$ and $|\underline{\hat{v}}_1| = |\underline{\hat{e}}_1|$. Clearly, $|\underline{\hat{e}}| \hookrightarrow \langle |\underline{\hat{e}}_1|, e_2' \rangle$ for some e_2' such that $|\underline{\hat{e}}_2| \hookrightarrow e_2'$ holds. By induction hypothesis, $\underline{\hat{e}}_2 \hookrightarrow^+ \underline{\hat{e}}_2'$ for some $\underline{\hat{e}}_2'$ such that $|\underline{\hat{e}}_2'| = e_2'$. Let $\underline{\hat{e}}' = (\langle \underline{\hat{v}}_1, \underline{\hat{e}}_2' \rangle)^1$, and we have $\underline{\hat{e}} \hookrightarrow^+ \underline{\hat{e}}'$ and $|\underline{\hat{e}}'| = \langle |\underline{\hat{v}}_1|, |\underline{\hat{e}}_2'| \rangle = \langle |\underline{\hat{e}}_1|, e_2' \rangle$.
  - $|\underline{\hat{e}}_1|$ is not a value. Then $|\underline{\hat{e}}| \hookrightarrow \langle e_1', |\underline{\hat{e}}_2| \rangle$ for some e_1' such that $|\underline{\hat{e}}_1| \hookrightarrow e_1'$. By induction hypothesis, $\underline{\hat{e}}_1 \hookrightarrow^+ \underline{\hat{e}}_1'$ for some $\underline{\hat{e}}_1'$ such that $|\underline{\hat{e}}_1'| = e_1'$. Let $\underline{\hat{e}}' = (\langle \underline{\hat{e}}_1', \underline{\hat{e}}_2 \rangle)^1$, and we are done.

The rest of the cases can be handled similarly.

By Theorem 4 and Theorem 5, we know that the erasure of a well-type a-expression $\underline{\hat{e}}$ preserves the dynamic semantics of $\underline{\hat{e}}$ w.r.t. the erasure function. The attributive types in $\mathcal{L}_0$ are precisely introduced to establish this property.

$$\frac{\Sigma \vdash \tau_1 : type_{b_1} \quad \Sigma \vdash \tau_2 : type_{b_2}}{\Sigma \vdash \tau_1 * \tau_2 : type_{b_1 \oplus b_2}} \ \textbf{(srt-tup)}$$

$$\frac{\Sigma \vdash \tau_1 : type_{b_1} \quad \Sigma \vdash \tau_2 : type_{b_2} \quad t \leq b_2}{\Sigma \vdash \tau_1 \rightarrow^{t} \tau_2 : type_{b_2}} \ \textbf{(srt-fun)}$$

$$\frac{\Sigma \vdash B : bool \quad \Sigma \vdash \tau : type_b}{\Sigma \vdash B \supset \tau : type_b} \ \textbf{(srt-}\supset\textbf{)} \qquad \frac{\Sigma \vdash B : bool \quad \Sigma \vdash \tau : type_b}{\Sigma \vdash B \wedge \tau : type_b} \ \textbf{(srt-}\wedge\textbf{)}$$

$$\frac{\Sigma, a : \sigma \vdash \tau : type_b}{\Sigma \vdash \forall a : \sigma.\ \tau : type_b} \ \textbf{(srt-}\forall\textbf{)} \qquad \frac{\Sigma, a : \sigma \vdash \tau : type_b}{\Sigma \vdash \exists a : \sigma.\ \tau : type_b} \ \textbf{(srt-}\exists\textbf{)}$$

Fig. 6. Some sorting rules for $\mathcal{L}_0^{\forall,\exists}$

3 Extension

The type system of $\mathcal{L}_0$, which is based on simple types, is not expressive enough to support interesting and realistic style of programming with theorem proving. In this section, we mention an extension of $\mathcal{L}_0$ to $\mathcal{L}_0^{\forall,\exists}$ with dependent types as well as polymorphic types. Formalizing an extension as such is just a common routine in the framework of Applied Type System [Xi04], and we have coined a word *predicatization* to refer to such a routine. Please see [Xi04,CX05] for more details that are not presented here for the sake of brevity.

The language $\mathcal{L}_0^{\forall,\exists}$ consists of a static component (statics) and a dynamic component (dynamics). The statics itself is a simply typed language and a type in it is referred to as a *sort*. We assume the existence of the following basic sorts: *bool*, *int*, $type_0$ and $type_1$, and we may write *prop* and *type* for $type_0$ and $type_1$, respectively; *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 and s for static terms, and write $\Sigma \vdash s : \sigma$ to mean that s can be given the sort σ under the context Σ. The additional forms of types in $\mathcal{L}_0^{\forall,\exists}$ (over those in $\mathcal{L}_0$) are given below:

$$\text{types } \tau ::= \ldots \mid \delta(\overline{s}) \mid B \supset \tau \mid B \wedge \tau \mid \forall a : \sigma.\ \tau \mid \exists a : \sigma.\ \tau$$

where $\overline{s}$ stands for a sequence of static terms. We use $B \supset T$ for a guarded type and $B \wedge T$ for an asserting type, where B and T refer to static expressions of sorts *bool* and *type*, respectively. As an example, the following type is for a function from natural numbers to negative integers:

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

where $\mathbf{int}(I)$ is a singleton type for the integer equal to I. 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.

Some of the rules for assigning sorts to static terms are given in Figure 6. In addition, we have the following sorting rule:

$$\frac{\Sigma \vdash \tau : \mathit{type}_1}{\Sigma \vdash \tau : \mathit{type}_0}$$

which allows a type to be used as a prop. It is this simple rule that initiates the study on attributive types.

Let us use a judgment of the form $\Sigma \vdash \tau : \mathit{type}_b \Rightarrow \underline{\hat{\tau}}$, where $\mathit{bit}(\underline{\hat{\tau}}) = b$ is assumed, to mean that $\Sigma \vdash \tau : \mathit{type}_b$ is derivable and τ is transformed into $\underline{\hat{\tau}}$ based the derivation of $\Sigma \vdash \tau : \mathit{type}_b$. Then we can readily turn the rules in Figure 6 into the ones for deriving judgments of this form. For instance, the following rule is derived from the rule **(srt-fun)**:

$$\frac{\Sigma \vdash \tau_1 : \mathit{type}_{b_1} \Rightarrow \underline{\hat{\tau}}_1 \quad \Sigma \vdash \tau_2 : \mathit{type}_{b_2} \Rightarrow \underline{\hat{\tau}}_2 \quad t \leq b_2}{\Sigma \vdash \tau_1 \rightarrow^t \tau_2 : \mathit{type}_{b_2} \Rightarrow (\underline{\hat{\tau}}_1 \rightarrow^t \underline{\hat{\tau}}_2)^{b_2}}$$

It should be obvious to see how the other rules in Figure 6 are handled, and we leave out the details. Note that using the sorting rules to attach erasure bits to types is particularly important in practice as requiring the programmer to do so *manually* would seem too unwieldy if not completely impractical..

A static term s in $\mathcal{L}_0^{\forall,\exists}$ is either a static boolean term B of sort *bool*, or a static integer I of sort *int*, or a prop P of sort *prop*, or a type T of 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 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 $\Theta : \Sigma$ (meaning $\Theta(a)$ can be assigned the sort $\Sigma(a)$ for every $a \in \mathbf{dom}(\Theta) = \mathbf{dom}(\Sigma)$), if each static boolean term in $\overline{B}[\Theta]$ equals *true* then so does $B[\Theta]$. In practice, such a constraint relation is often determined by some automatic decision procedure.

A typing judgment in $\mathcal{L}_0^{\forall,\exists}$ is of the form $\Sigma; \overline{B}; \Gamma \vdash \underline{\hat{e}} : \underline{\hat{\tau}}$, and we omit the typing rules for $\mathcal{L}_0^{\forall,\exists}$. The developed theory in Section 2 can be readily carried over to $\mathcal{L}_0^{\forall,\exists}$. To establish program (or proof) termination more effectively in practice, we employ an approach that allows the programmer to supply *termination metrics* for automatic termination verification [Xi02]. We will explain some uses of this approach in Section 4.

4 An Example

We now use an example to illustrate how the form of attributive types developed in this paper can be used to support programming with theorem proving.

In Figure 7, we declare a type constructor **tree** that takes a type T and two integers I_1 and I_2 to form the type $\mathbf{tree}(T, I_1, I_2)$ for binary trees of height I_1 and size I_2 in which each element is of type T. We use $\mathit{max}(h_1, h_2)$ for the

```
datatype tree (type, int, int) =
  | {a:type} E (a, 0, 0)
  | {a:type} {h1,h2,s1,s2:nat}
    B (a, 1+max(h1,h2), 1+s1+s2) of (tree (a,h1,s1), a, tree (a,h2,s2))

dataprop POW2 (int, int) = // POW2 (p, n) means 2^p = n
  | POW2bas (0, 1)
  | {p,n:nat} POW2ind (p+1, n+n) of POW2 (p, n)
```

Fig. 7. A dependent datatype for binary trees and a dataprop for encoding power of 2

maximum of h_1 and h_2. Clearly, for any binary tree of height I_1 and size I_2, we have $2^{I_1} < I_2$. To establish this property, we declare a prop constructor $\underline{\mathbf{POW2}}$ in Figure 7 to encode the power function with base 2: Given integers I_1 and I_2, if $\underline{\mathbf{POW2}}(I_1, I_2)$ is inhabited, the $2^{I_1} = I_2$ holds.

```
// [pow2] computes powers of two
fun pow2 {p:nat} .<p>. (p: int p): [n:nat] (POW2 (p, n) | int n) =
  if p igt 0 then let
    val (pf | n) = pow2 (ipred p)
  in
    (POW2ind pf | n iadd n)
  end else begin
    (POW2bas | 1)
  end

// [prfun] means that a proof function is declared; instead of
// of type, a prop is assigned to a proof function
prfun lemma {a:type} {h,s:nat} .<h>.
  (t: tree (a, h, s)): [n: nat | s < n] POW2 (h, n) = case+ t of
  | E () => POW2bas ()
  | B (t1, _, t2) => let
      prval pf1 = lemma t1 and pf2 = lemma t2
      val (pf | _) = pow2 (ipred (height t))
      prval () = pow2_inc (pf1, pf) and () = pow2_inc (pf2, pf)
    in
      POW2ind pf
    end
```

Fig. 8. A proof construction involving total functions

We implement a function *pow2* and a proof function *lemma* in Figure 8, which are assigned the following type and prop, respectively:

$$\begin{array}{rcl} \mathit{pow2} & : & \forall \iota_1.\ \iota_1 \geq 0 \supset (\mathbf{int}(\iota_1) \to^0 \exists \iota_2.\ \underline{\mathbf{POW2}}(\iota_1, \iota_2) * \mathbf{int}(\iota_2)) \\ \mathit{lemma} & : & \forall \alpha. \forall \iota_1. \forall \iota_2.\ \mathbf{tree}(\alpha, \iota_1, \iota_2) \to^0 \exists \iota.\ (\iota_2 < \iota) \wedge \underline{\mathbf{POW2}}(\iota_1, \iota) \end{array}$$

Note that given an integer I, we use $\mathbf{int}(I)$ for the singleton type containing the only integer of value I. Clearly, this prop means that $I_2 < 2^{I_1}$ if there exists a tree of height I_1 and size I_2. In the definition of and *pow2* and *lemma*, the functions *iadd*, *ipred*, *igt* and *height*, and the proof function *pow2_inc* are assigned the following types and prop:

$$
\begin{array}{rcl}
\mathit{iadd} & : & \forall\iota_1\forall\iota_2.\ (\mathbf{int}(\iota_1) * \mathbf{int}(\iota_2)) \to^0 \mathbf{int}(\iota_1 + \iota_2) \\
\mathit{ipred} & : & \forall\iota.\ \mathbf{int}(\iota) \to^0 \mathbf{int}(\iota - 1) \\
\mathit{igt} & : & \forall\iota_1\forall\iota_2.\ (\mathbf{int}(\iota_1) * \mathbf{int}(\iota_2)) \to^0 \mathbf{bool}(\iota_1 > \iota_2) \\
\mathit{height} & : & \forall\alpha.\forall\iota_1.\forall\iota_2.\ (\iota_1 \geq 0 \wedge \iota_2 \geq 0) \supset \mathbf{tree}(\alpha, \iota_1, \iota_2) \to^0 \mathbf{int}(\iota_1) \\
\mathit{pow2\_inc} & : & \forall\iota_1.\forall\iota_1'.\forall\iota_2.\forall\iota_2'.\ (\iota_1 \geq 0 \wedge \iota_2 \geq 0 \wedge \iota_1 \leq \iota_2) \supset \\
& & \quad (\underline{\mathbf{POW2}}(\iota_1, \iota_1') * \underline{\mathbf{POW2}}(\iota_2, \iota_2') \to^0 (\iota_1' \leq \iota_2') \wedge \mathbf{1})
\end{array}
$$

Note that we use $\mathbf{bool}(B)$ for the singleton type containing the only boolean of value B. The functions *iadd*, *ipred* and *igt* are primitive ones with the obvious meaning. The function *height* computes the height of a given tree and the proof function *pow2_inc* essentially proves that $2^{I_1} \leq 2^{I_2}$ holds for any natural number I_1 and I_2 satisfying $I_1 \leq I_2$. For brevity, we omit the actual code that implements *height* and *pow2_inc*.

5 Related Work and Conclusion

In an attempt to advance the type system of ML, Dependent ML (DML) is developed to support a restricted form of dependent types where type indexes are required to be drawn only from a chosen index language [Xi07,XP99]. In DML, type-checking a sufficiently annotated program can be reduced to solving constraints from this chosen index language, which is often handled through a fully automatic but limited decision procedure. In ATS [Xi], a paradigm of programming with theorem proving is introduced [CX05], making it possible for the programmer to handle (difficult) constraints by constructing explicit proofs. Consequently, the need for proof erasure appears.

The approach to proof erasure in this paper draws primary inspiration from the design of program extraction (from proofs) in Coq [PM89,Let03]. In particular, the sorts *prop* and *type* here roughly corresponds to the kinds *Prop* and *Set* in Coq.

The notion of proof erasure in this paper is also casually related to a modal type theoretical study on proof irrelevance [Pfe01]. The approach taken in [Pfe01] is fundamentally different from the one in [PM89] as the former does not support, *a priori*, a separation between props and types. Instead, whether an object can be classified as a program (i.e., intentional expression) or a proof only depends on some conditions on its free variables. We, however, do not make use of such conditions when formulating the typing rules for $\mathcal{L}_0$.

References

[BC04] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the NuPrl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986. ISBN 0-13-451832-2. x+299 pp.

[CX05] Chiyan Chen and Hongwei Xi. Combining Programming with Theorem Proving. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 66–77. Tallinn, Estonia, September 2005.

[DE73] G.B. Dantzig and B.C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[DFH+93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Technique 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[JG91] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Proceedings of 18th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 303–310, January 1991.

[Let03] Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *post-workshop Proceedings of TYPES 2002*. Springer-Verlag LNCS 2646, The Netherlands, 2003.

[Mar84] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Naples, Italy, 1984. ISBN 88-7088-105-9. ix+91 pp.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *International Series of Monographs on Computer Science*. Clarendon Press, Oxford, 1990. ISBN 0-19-853814-6. x+221 pp.

[Pfe01] Frank Pfenning. Intentionality, Extentionality and Proof Irrelevance in Modal Type Theory. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, pages 221–230. Boston, June 2001.

[PM89] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse de doctorat, Université de Paris VII, Paris, France, 1989.

[Xi] Hongwei Xi. The ATS Programming Language. Available at: `http://www.ats-lang.org/`.

[Xi02] Hongwei Xi. Dependent Types for Program Termination Verification. *Journal of Higher-Order and Symbolic Computation*, 15(1):91–132, March 2002.

[Xi04] Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.

[Xi07] Hongwei Xi. Dependent ML: an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.

[XP99] Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *Proceedings of 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227. ACM press, San Antonio, Texas, January 1999.