ATS/LF: A type system for constructing proofs as total functional programs[1]

HONGWEI XI

1 Introduction

The development of *Applied Type System* ($\mathcal{ATS}$) [36, 31] stems from an earlier attempt to introduce dependent types into practical programming [38, 37]. While there is already a framework *Pure Type System* [4] ($\mathcal{PTS}$) that offers a simple and general approach to designing and formalizing type systems, it has been understood that there are some acute problems with $\mathcal{PTS}$ that make it difficult to support, especially, in the presence of dependent types various common programming features such as general recursion [10], recursive types [19], effects [17], exceptions [15] and input/output, etc. To address such limitations of $\mathcal{PTS}$, $\mathcal{ATS}$ is proposed to allow for designing and formalizing type systems that can readily accommodate common realistic programming features. The key salient feature of $\mathcal{ATS}$ lies in a complete separation of the statics, where types are formed and reasoned about, from the dynamics, where programs are constructed and evaluated. With this separation, it is no longer possible for a program to occur in a type as is otherwise allowed in $\mathcal{PTS}$.

We have now designed and implemented ATS, a programming language with its type system rooted in $\mathcal{ATS}$. The work we report here is primarily motivated by a need for combining programs with proofs in ATS. Before going into further details, we would like to present an example to clearly illustrate the motivation. In ATS, we can declare a function *append* (through a form of syntax rather similar to that of Standard ML [20]) in Figure 1. We use **list** as a type constructor. When applied to a type T and an integer I, **list**(T, I) forms a type for lists of length I in which each element is of type T. The two list constructors *nil* and *cons* are assigned the following

```
fun append {a:type} {m,n:nat}
  (xs: list (a, m), ys: list (a, n)): list (a, m+n) =
  case xs of
  | nil () => ys  // 1st clause
  | cons (x, xs) => cons (x, append (xs, ys)) // 2nd clause

dataprop MUL (int, int, int) =
  | {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)

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

Figure 1. An example of proof construction in ATS/LF

constant types (or c-types):

$$nil \quad : \quad \forall a : type. \ \mathbf{list}(a, 0)$$
$$cons \quad : \quad \forall a : type. \forall n : nat. \ (a, \mathbf{list}(a, n)) \Rightarrow \mathbf{list}(a, n+1)$$

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

$$\forall a : type. \forall m : nat. \forall n : nat. \ (\mathbf{list}(a, m), \mathbf{list}(a, n)) \rightarrow \mathbf{list}(a, m+n)$$

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 ATS, and a static term of sort *type* is a type (for dynamic terms). Also, *int* is a built-in sort for integers in ATS, and *nat* is the subset sort $\{a : int \mid a \geq 0\}$ for natural numbers. We will explain later that subset sorts are just of a form of syntactic sugar.

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

$$\forall m : nat. \forall n : nat. \ m = 0 \supset n = m + n$$
$$\forall m : nat. \forall n : nat. \forall m' : int. \ m = m' + 1 \supset (m' + n) + 1 = m + n$$

The first constraint is generated when the first clause in the definition of *append* is type-checked; the constraint 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 in the definition of *append* is type-checked; the constraint is needed for determining that for any integer m', 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 such constraints automatically. 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 [12] to solve such constraints. While this is indeed a simple design, it is inherently *ad hoc* and can also be too restrictive, sometimes, in a situation where nonlinear constraints need to be handled. For instance, a function *concat* that concatenates m lists of length n may be given the following type:

$$\forall a : type.\forall m : nat.\forall n : nat.\ \mathbf{list}(\mathbf{list}(a, n), m) \rightarrow \mathbf{list}(a, m \times n)$$

Unfortunately, this type is not allowed in ATS as $m \times n$ is not a linear arithmetic expression and thus cannot be used as a type index. To address this issue, a recursive dependent *prop* constructor $\mathbf{MUL}$ is declared in Figure 1 for encoding the multiplication function on integers. In general, a prop is like a type, and the essential difference between them is that a prop can only be assigned to total terms (to be formally defined later), which we often refer to as *proof terms*. The concrete syntax used to declare $\mathbf{MUL}$ indicates that there are three (proof) value constructors associated with $\mathbf{MUL}$, which are given the following constant props (or c-props):

$$
\begin{aligned}
MULbas \quad &: \quad \forall n : int.\ () \Rightarrow \mathbf{MUL}(0, n, 0) \\
MULind \quad &: \quad \forall m : int.\forall n : int.\forall p : int.\ m \geq 0 \supset \\
&\qquad (\mathbf{MUL}(m, n, p) \Rightarrow \mathbf{MUL}(m + 1, n, p + n)) \\
MULneg \quad &: \quad \forall m : int.\forall n : int.\forall p : int.\ m > 0 \supset \\
&\qquad (\mathbf{MUL}(m, n, p) \Rightarrow \mathbf{MUL}(-m, n, -p))
\end{aligned}
$$

Given integers I_1, I_2 and I_3, it is clear that $I_1 \times I_2 = I_3$ holds if and only if $\mathbf{MUL}(I_1, I_2, I_3)$ 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:

$$
\begin{aligned}
0 \times n \quad &= \quad 0; \\
(m + 1) \times n \quad &= \quad m \times n + n \ \text{ if } m \geq 0; \\
(-m) \times n \quad &= \quad -(m \times n) \ \text{ if } m > 0.
\end{aligned}
$$

---

[1]More precisely, each arithmetic constraint is required to be turned into a linear programming problem.

The function *concat* can now be given the following type:

$$\forall a : type.\forall m : nat.\forall n : nat.$$
$$\mathbf{list}(\mathbf{list}(a, n), m) \rightarrow \exists p : nat.\ \mathbf{MUL}(m, n, p) * \mathbf{list}(a, p)$$

The code for implementing *concat* is given in Figure 1. We write $(\ldots)$ to form tuples. Also, we use the bar symbol $(\,|\,)$ as a separator to separate proofs from programs. 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 $\mathbf{MUL}(I_1, I_2, I_3)$ 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* equals $I_1 \times I_2$.[2] Now suppose we would like to assign *concat* the following type:

$$\forall a : type.\forall m : nat.\forall n : nat.$$
$$\mathbf{list}(\mathbf{list}(a, n), m) \rightarrow \exists p : nat.\ \mathbf{MUL}(n, m, p) * \mathbf{list}(a, p)$$

which is obtained from replacing $\mathbf{MUL}(m, n, p)$ with $\mathbf{MUL}(n, m, p)$ in the above type assigned to *concat*. Then we need to replace *MULbas*() and *MULind*(*pf*) in the definition of *concat* with *lemma0*() and *lemma1*(*pf*), respectively, where *lemma0* and *lemma1* are the following (proof) functions:

```
prfun lemma0 {n:nat} .<n>. (): MUL (n, 0, 0) =
  // [sif] forms a conditional with a static condition
  sif n > 0 then MULind (lemma0 {n-1} ()) else MULbas ()

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

Note that the keyword *prfun* indicates the implementation of a proof function. We now choose *lemma1* for further explanation. The prop assigned to *lemma1* is

$$\forall m : nat.\forall n : nat.\forall p : int.\ \mathbf{MUL}(n, m, p) \rightarrow \mathbf{MUL}(n, m + 1, p + n)$$

Essentially, *lemma1* represents an inductive proof of $n \times m = p \supset n \times (m + 1) = p + n$ 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 *lemma1* are type-checked:

$$\forall n : nat.\forall p : int.\ n = 0 \supset (p = 0 \supset 0 = p + n)$$
$$\forall m : nat.\forall n : nat.\forall p : int.\forall n' : int.\forall p' : int.$$
$$n = n' + 1 \supset (p = p' + m \supset p + n = (p' + n') + (m + 1))$$

---

[2]However, there is really no need for constructing proof values like *pf* at run-time, and this issue is already investigated elsewhere [6].

In order for *lemma1* to represent a proof, we need to show that *lemma1* is a total function, that is, given *pf* of prop $\mathbf{MUL}(I_2, I_1, I_3)$ for natural numbers I_1 and I_2 and integer I_3, *lemma1*(*pf*) is guaranteed to return a proof value of prop $\mathbf{MUL}(I_2, I_1 + 1, I_3 + I_2)$. In this paper, we are to present a type system *ATS/LF* in which every well-typed function is guaranteed to be total. Generally speaking, when implementing a recursive function in *ATS/LF*, the programmer is required to provide a metric that can be used to verify the termination of the function. For instance, in the definition of *lemma1*, $\langle n \rangle$ is the provided metric for verifying that *lemma1* is terminating; when *lemma1* is applied to a value of prop $\mathbf{MUL}(I_2, I_1, I_3)$, the label $\langle I_2 \rangle$ is associated with this call; in the case where a recursive call to *lemma1* is made subsequently, the label associated with the recursive call is $\langle I_2 - 1 \rangle$ (since *pf'* in the definition of *lemma1* is given the type $\mathbf{MUL}(I_2 - 1, I_1, I_3 - I_1)$), which is strictly less than the label $\langle I_2 \rangle$ associated with the original call; as a label associated with *lemma1* is always a natural number, it is evident that *lemma1* is terminating. To show that *lemma1* is total, we also need to verify that pattern matching in the definition of *lemma1* can never fail, which is a topic already studied elsewhere [33, 35].

The primary motivation for developing *ATS/LF* is to support in ATS a programming paradigm that combines programs with proofs. For brevity, we, however, are unable to formally demonstrate in this paper as to how such a combination can take place, and we refer the interested reader to [6] for further details on this subject. Instead, we focus on the formalization of *ATS/LF*, establishing that every-well typed program in *ATS/LF* is total. A secondary motivation we have is to use *ATS/LF* as a logical framework for encoding deduction systems and their properties, and we are to present some examples in support of such an application of *ATS/LF*.

The rest of the paper is organized as follows. In Section 2, we first mention some closely related work. We then formalize *ATS/LF* in Section 3. In particular, we make use of the notion of reducibility [30] in proving that every well-typed program in *ATS/LF* is total. In support of using *ATS/LF* as a (meta) logical framework, we demonstrate how deduction systems can be encoded in *ATS/LF* by presenting some interesting examples in Section 5, which are all verified in ATS. We conclude in Section 6.

2 Related Work

The approach to termination verification in *ATS/LF* is essentially taken from an earlier work [34], where a notion of termination metrics is introduced into Dependent ML (DML) [38, 37] to support termination verification for programs in DML. In this paper, we give an account for this approach in a more general setting (e.g., functional type indexes, which are

not allowed in DML, are supported in ATS/LF).

There is certainly a vast body of literature on termination verification. In type theory, a standard approach to proving termination (of functions or programs) derives from the notion of accessible predicate [1, 21], and a detailed study based on it can be found in [5].

When used as a logical framework, ATS/LF is probably most closely related to Twelf [27]. In particular, a dataprop declaration in ATS/LF corresponds to a declaration for a type constructor in Twelf plus the constants associated with the type constructor. The approach to termination verification (for logical programs) in Twelf [28] is similar to that of ATS/LF in the aspect that it requires a structural ordering (possibly on higher-order terms) to be provided by the user, though the justification for the correctness of this approach is not based on the notion of reducibility. In Twelf, a proof is really a meta concept and it can not be represented within Twelf while in ATS/LF, a proof is just a well-typed program. This is a fundamental difference between Twelf and ATS/LF, which greatly influences the manner in which deduction systems are encoded. Recently, Delphin, a functional programming language built on top of Twelf, is proposed [29] and termination proofs in Twelf are expected to be represented as total functional programs in Delphin.

Of course, the related work also includes various (interactive) theorem proving systems based on type theory such as NuPrl [9], Coq [14] and Isabelle [23]. In order to effectively reason about program properties within a type theory, the underlying functional language of a theorem proving system is often required to be relatively pure, making it difficult to support many realistic programming features (e.g., general recursion, references, exceptions). In general, it is inflexible as well as involved to construct programs in a theorem proving system, though significant progress has been made in this direction. In contrast, ATS/LF is primarily designed to be part of ATS, a programming language that can readily support realistic programming features (e.g., pointers and pointer arithmetic [39]). In this respect, the design of ATS/LF is unique, and it has not been seen elsewhere in the literature.

3   Formal Development

In this section, we formally present ATS/LF, a type system rooted in the framework $\mathcal{ATS}$ that can guarantee the totality of every-well typed program. There are two components in ATS/LF: the static component (statics) and the dynamic component (dynamics). We first give the syntax for the statics

$$\frac{\Sigma(a) = \sigma}{\Sigma \vdash a : \sigma} \qquad \frac{\vdash sc : (\sigma_1, \ldots, \sigma_n) \Rightarrow \sigma \quad \Sigma \vdash s_i : \sigma_i \;\; \text{for } 1 \le i \le n}{\Sigma \vdash sc(s_1, \ldots, s_n) : \sigma}$$

$$\frac{\Sigma \vdash s_1 : \sigma_1 \quad \Sigma \vdash s_2 : \sigma_2}{\Sigma \vdash \langle s_1, s_2 \rangle : \sigma_1 * \sigma_2} \qquad \frac{\Sigma \vdash s : \sigma_1 * \sigma_2}{\Sigma \vdash \pi_1(s) : \sigma_1} \qquad \frac{\Sigma \vdash s : \sigma_1 * \sigma_2}{\Sigma \vdash \pi_2(s) : \sigma_2}$$

$$\frac{\Sigma, a : \sigma_1 \vdash s : \sigma_2}{\Sigma \vdash \lambda a : \sigma_1.s : \sigma_1 \to \sigma_2} \qquad \frac{\Sigma \vdash s_1 : \sigma_1 \to \sigma_2 \quad \Sigma \vdash s_2 : \sigma_1}{\Sigma \vdash s_1(s_2) : \sigma_2}$$

Figure 2. The rules for assigning sorts to static terms

as follows.

sorts	σ	::=	$b \mid \sigma_1 * \sigma_2 \mid \sigma_1 \to \sigma_2$
static terms	s	::=	$a \mid sc(\vec{s}) \mid$
			$\langle s_1, s_2 \rangle \mid \pi_1(s_1) \mid \pi_2(s_2) \mid$
			$\lambda a : \sigma.s \mid s_1(s_2)$
props	P	::=	$\delta(\vec{s}) \mid P_1 \to P_2 \mid B \supset P \mid$
			$\forall a : \sigma.P \mid B \wedge P \mid \exists a : \sigma.P$
static variable contexts	Σ	::=	$\emptyset \mid \Sigma, a : \sigma$
static substitutions	Θ	::=	$[] \mid \Theta[a \mapsto s]$

The statics itself is a simply typed language and a type in it is called *sort*. We use b for base sorts, σ for sorts, a for variables, and s for terms in the statics. There are certain constants sc in statics, which are either constructors scc or functions scf. Each sc is given a constant sort (c-sort) of the form $(\sigma_1, \ldots, \sigma_n) \Rightarrow \sigma$, indicating that $sc(s_1, \ldots, s_n)$ is a term of sort σ if s_i can be assigned sorts σ_i for $1 \le i \le n$, and we may write scc for $scc()$. Note that a c-sort is *not* considered a (regular) sort.

We use Σ for static variable contexts, which assign sorts to static variables, and $\mathbf{dom}(\Sigma)$ for the set of variables declared in Σ. We may write $\vec{a} : \vec{\sigma}$ for the static variable context $\emptyset, a_1 : \sigma_1, \ldots, a_n : \sigma_n$, where $\vec{a} = a_1, \ldots, a_n$ and $\vec{\sigma} = \sigma_1, \ldots, \sigma_n$. A sorting judgment is of the form $\Sigma \vdash s : \sigma$, which means that s can be assigned the sort σ under Σ. The rules for assigning sorts to terms are given in Figure 2. Also, we may write $\Sigma \vdash \vec{s} : \vec{\sigma}$ to mean that $\Sigma \vdash s_i : \sigma_i$ for $1 \le i \le n$, where $\vec{s} = s_1, \ldots, s_n$ and $\vec{\sigma} = \sigma_1, \ldots, \sigma_n$.

We use the names *static variable*, *static constant* and *static term* for a variable, a constant and a term in statics. A static substitution is a finite mapping from static variables to static terms. We use $[]$ for the empty mapping and $\Theta[a \mapsto s]$ for the mapping that extends Θ with a link from a to s. Also, we write $\bullet[\Theta]$ for the result of applying Θ to some syntax $\bullet$. Given Σ

$$\frac{\vdash \delta(\vec{\sigma}) \ [\mathbf{prop}] \quad \Sigma \vdash \vec{s} : \vec{\sigma}}{\Sigma \vdash \delta(\vec{s}) \ [\mathbf{prop}]} \qquad \frac{\Sigma \vdash P_1 \ [\mathbf{prop}] \quad \Sigma \vdash P_2 \ [\mathbf{prop}]}{\Sigma \vdash P_1 \to P_2 \ [\mathbf{prop}]}$$

$$\frac{\Sigma \vdash B : bool \quad \Sigma \vdash P \ [\mathbf{prop}]}{\Sigma \vdash B \supset P \ [\mathbf{prop}]} \qquad \frac{\Sigma, a : \sigma \vdash P \ [\mathbf{prop}]}{\Sigma \vdash \forall a : \sigma.P \ [\mathbf{prop}]}$$

$$\frac{\Sigma \vdash B : bool \quad \Sigma \vdash P \ [\mathbf{prop}]}{\Sigma \vdash B \wedge P \ [\mathbf{prop}]} \qquad \frac{\Sigma, a : \sigma \vdash P \ [\mathbf{prop}]}{\Sigma \vdash \exists a : \sigma.P \ [\mathbf{prop}]}$$

Figure 3. The rules for forming props

and Θ, we write $\Theta : \Sigma$ to mean that $\mathbf{dom}(\Theta) = \mathbf{dom}(\Sigma)$ and $\emptyset \vdash \Theta(a) : \Sigma(a)$ holds for each $a \in \mathbf{dom}(\Theta)$. In general, we write $\Sigma_1 \vdash \Theta : \Sigma_2$ to mean that $\Sigma_1 \vdash \Theta(a) : \Sigma_2(a)$ holds for each $a \in \mathbf{dom}(\Theta) = \mathbf{dom}(\Sigma_2)$.

We assume the existence of a base sort *bool* and the static constants *tt* and *ff* that are given the c-sort $() \Rightarrow bool$. Also, we may write B for static terms of sort *bool*. For each sort σ, we have a binary relation $=_\sigma$ of c-sort $(\sigma, \sigma) \Rightarrow bool$. In practice, we have a sort *int* for integers, and we also provide syntax for the programmer to declare new sorts, which is to be shown in the examples we present.

The rule for forming props are given in Figure 3. We write $\vdash \delta(\vec{\sigma}) \ [\mathbf{prop}]$ to mean that δ is prop constructor that takes static terms $\vec{s}$ of sorts $\vec{\sigma}$ to form a prop. As a convenient notation, we may write $\forall \Sigma$ for a sequence of quantifiers: $\forall a_1 : \sigma_1 \ldots \forall a_n : \sigma_n$, where $\Sigma = \emptyset, a_1 : \sigma_1, \ldots, a_n : \sigma_n$. We call the forms $B \supset P$ and $B \wedge P$ guarded props and asserting props, respectively, and we often employ subset sorts, a form of syntactic suguar, in the concrete syntax of ATS/LF when forming such props. For example, when writing $\forall a : nat.P$ ($\exists a : nat.P$), we really mean $\forall a : int. \ a \geq 0 \supset P$ ($\exists a : int. \ a \geq 0 \wedge P$). We now introduce the notion of constraints as follows.

DEFINITION 1. A constraint relation is of the form $\Sigma; \overline{B} \models B_0$, where $\overline{B}$ stands for a sequence $B_1, \ldots, B_n$ such that $\Sigma \vdash B_i : bool$ holds for each $0 \leq i \leq n$. A constraint $\Sigma; \overline{B} \models B_0$ is satisfied if for each substitution $\Theta : \Sigma$, $\overline{B}[\Theta]$ contains *ff* or $B_0[\Theta]$ is *tt*. In addition, we write $\Sigma; \overline{B} \models \overline{B}_0$ to mean that $\Sigma; \overline{B} \models B_0$ holds for each B_0 in $\overline{B}_0$.

A proper approach to defining constraint relation is through the use of models for type theory [16, 2]. This is done, for instance, in [31], where some techniques in [3] are employed to construct a model for the statics of a generic applied type system. Also, a detailed construction of such a

$$\frac{\Sigma; \overline{B} \models s_i =_\sigma s'_i}{\Sigma; \overline{B} \models \delta(s_1, \ldots, s_n) \leq_{pr} \delta(s'_1, \ldots, s'_n)} \qquad \frac{\Sigma; \overline{B} \models P'_1 \leq_{pr} P_1 \quad \Sigma; \overline{B} \models P_2 \leq_{pr} P'_2}{\Sigma; \overline{B} \models P_1 \to P_2 \leq_{pr} P'_1 \to P'_2}$$

$$\frac{\Sigma; \overline{B}, B' \models B \quad \Sigma; \overline{B}, B' \models P \leq_{pr} P'}{\Sigma; \overline{B} \models B \supset P \leq_{pr} B' \supset P'} \qquad \frac{\Sigma, a : \sigma; \overline{B} \models P \leq_{pr} P'}{\Sigma; \overline{B} \models \forall a : \sigma.P \leq_{pr} \forall a : \sigma.P'}$$

$$\frac{\Sigma; \overline{B}, B \models B' \quad \Sigma; \overline{B}, B \models P \leq_{pr} P'}{\Sigma; \overline{B} \models B \wedge P \leq_{pr} B' \wedge P'} \qquad \frac{\Sigma, a : \sigma; \overline{B} \models P \leq_{pr} P'}{\Sigma; \overline{B} \models \exists a : \sigma.P \leq_{pr} \exists a : \sigma.P'}$$

Figure 4. The subprop rules for *ATS/LF*

$$
\begin{array}{llll}
\text{dynamic terms} & d & ::= & x \mid f \mid dc[\vec{s}](\vec{d}) \mid \mathbf{lam}\, x.d \mid \mathbf{app}(d_1, d_2) \mid \\
& & & \mathbf{fix}\, f[\vec{a}].m \Rightarrow d \mid \mathbf{sif}(s, d_1, d_2) \mid \\
& & & \supset^+(d) \mid \supset^-(d) \mid \mathbf{lam}\, \vec{a}.d \mid \mathbf{app}(d, \vec{s}) \mid \\
& & & \wedge(d) \mid \mathbf{let}\, \wedge(x) = d_1 \,\mathbf{in}\, d_2 \mid \\
& & & \langle s, d \rangle \mid \mathbf{let}\, \langle a, x \rangle = d_1 \,\mathbf{in}\, d_2 \mid \\
\text{dynamic values} & v & ::= & x \mid dcc[\vec{s}](\vec{v}) \mid \mathbf{lam}\, x.d \mid \\
& & & \supset^+(d) \mid \mathbf{lam}\, \vec{a}.d \mid \wedge(v) \mid \langle s, v \rangle \\
\text{dynamic var. ctx.} & \Pi & ::= & \emptyset \mid \Pi, x : P \mid \Pi, f : \forall \Sigma.m \Rightarrow P \\
\text{labeling} & \mu & ::= & \emptyset \mid \mu, f : s \\
\text{dynamic subst.} & \theta & ::= & [] \mid \theta[xf \mapsto d]
\end{array}
$$

Figure 5. The syntax for the dynamics of *ATS/LF*

model can be found in [37]. In practice, we often impose certain restrictions on the constraint relation so that an effective means can be found to solve constraints automatically. For instance, in our current implementation of ATS [32], the imposed restrictions guarantee that each (arithmetic) constraint can be turned into a problem in linear integer programming.

Given two props P and P', we write $P \leq_{pr} P'$ to mean that P is a subprop of P'. A subprop judgment is of the form $\Sigma; \overline{B} \models P \leq_{pr} P'$, and the rules for deriving such judgments are given in Figure 4. Note that a subprop judgment $\Sigma; \overline{B} \models P \leq_{pr} P'$ is *conditional* in the sense that $P \leq_{pr} P'$ is decided under the conditions $\overline{B}$. This is a rather powerful notion in $\mathcal{ATS}$. In contrast, type equality in Martin-Löf's constructive type theory [18, 22] and related systems such as construction of calculus [11] are unconditional in the sense that two types are considered equal if and only if they are $\beta\eta$-equivalent.

The syntax for the dynamics of ATS/LF is given in Figure 5. We use x for a lam-variable and f for a fix-variable, and xf for a dynamic variable, which is either an x or an f. We use dc for dynamic constants, which are either dynamic constructors dcc or dynamic functions dcf. We assume that each dynamic constant dc is assigned a constant prop (c-prop) of the form $\forall \Sigma.\overline{B} \supset ((P_1, \ldots, P_n) \Rightarrow P)$. Note that a c-prop is not considered a (regular) prop. A dynamic variable context Π assigns props to lam-variables and decorated props, which are of the form $\forall \Sigma.m \Rightarrow P$, to fix-variables. The construct **sif** forms a conditional expression where the condition is a static term. A labeling μ associates static terms with fix-variables, and a dynamic substitution maps dynamic variables to dynamic terms. In ATS/LF, we often use the name *proof term* and *proof value* to refer to a dynamic term and a dynamic value, respectively.

We now introduce termination metrics as follows, which play a key rôle in guaranteeing that each well-typed program in ATS/LF is terminating.

DEFINITION 2. Given a sort σ, a binary transitive relation $<$ on static terms s of the sort σ is well-founded if there are no infinitely many s_i of the sort σ such that $s_{i+1} < s_i$ hold for all $i \geq 0$. For instance, a common well-founded ordering we use is the lexicographic ordering on tuples of natural numbers. Given a well-founded binary relation $<$ on static terms of sorts σ and a static term s of sort σ, $(<, s)$ is a metric. Note that s may contain free static variables. We use m for metrics.

A judgment for assigning a prop to a dynamic term in ATS/LF is of the form $\Sigma; \overline{B}; \Pi \vdash d : P \ll \mu$. The rules for assigning props to dynamic terms in ATS/LF are listed in Figure 6. Note that the obvious side conditions associated with certain rules are omitted. In the case where μ is empty, we may write $\Sigma; \overline{B}; \Pi \vdash d : P$ for $\Sigma; \overline{B}; \Pi \vdash d : P \ll \mu$. The rule **(pr-fix-var)** indicates that each occurrence of a fix-variable f in a dynamic term must be inside a term of the form $\mathbf{app}(f, \vec{s})$; if f is assigned a decorated prop $\forall \vec{a} : \vec{\sigma}.m \Rightarrow P_0$ for $m = (<, s_0)$, then we say that a label $s_0[\vec{a} \mapsto \vec{s}]$ is attached to this occurrence of f. A judgment of the form $\Sigma; \overline{B}; \Pi \vdash d : P \ll \mu$ basically means that d can be assigned the prop P under $\Sigma; \overline{B}; \Pi$; in addition, given any fix-variable f such that $\Sigma(f) = \forall \vec{a} : \vec{\sigma}.m \Rightarrow P_0$ for $m = (<, s_0)$ and $\mu(f) = s_1$ (i.e., $f : s_1$ occurs in μ), the label attached to each occurrence of f in d is strictly less than s_1 (according to the ordering $<$).

As usual, we have the following substitution lemma in ATS/LF.

LEMMA 3 (Substitution). *Assume that $\Sigma; \overline{B}; \Pi \vdash d : P \ll \mu$ is derivable.*

1. *If $\Sigma = \Sigma', \Sigma''$ and $\Sigma' \vdash \Theta : \Sigma''$ is derivable, then $\Sigma'; \overline{B}[\Theta]; \Pi[\Theta] \vdash d[\Theta] : P[\Theta] \ll \mu[\Theta]$ is also derivable.*

$$\frac{\Sigma; \overline{B}; \Pi \vdash d : P \quad \Sigma; \overline{B} \models P \leq_{pr} P'}{\Sigma; \overline{B}; \Pi \vdash d : P'} \text{ (pr-sub)}$$

$$\frac{\Pi(x) = P}{\Sigma; \overline{B}; \Pi \vdash x : P \ll \mu} \text{ (pr-lam-var)}$$

$$\frac{\Pi(f) = \forall \Sigma_0. \ (<, s_0) \Rightarrow P \quad \Sigma \vdash \Theta : \Sigma_0 \quad \Sigma; \overline{B} \models s_0[\Theta] < \mu(f)}{\Sigma; \overline{B}; \Pi \vdash \mathbf{app}(f, \vec{s}) : P \ll \mu} \text{ (pr-fix-var)}$$

$$\frac{\vdash dc : \forall \vec{a} : \vec{\sigma}.\overline{B}_0 \supset ((P_1, \ldots, P_n) \Rightarrow P) \quad \Sigma \vdash \vec{s} : \vec{\sigma} \quad \Sigma \models \overline{B}_0[\vec{a} \mapsto \vec{s}]}{\Sigma; \overline{B}; \Pi \vdash d_1 : P_1[\vec{a} \mapsto \vec{s}] \quad \ldots \quad \Sigma; \overline{B}; \Pi \vdash d_n : P_n[\vec{a} \mapsto \vec{s}]}{\Sigma; \overline{B}; \Pi \vdash dc[\vec{s}](d_1, \ldots, d_n) : P[\vec{a} \mapsto \vec{s}]} \text{ (pr-const)}$$

$$\frac{\Sigma; \overline{B}; \Pi, x : P_1 \vdash d : P_2 \ll \mu}{\Sigma; \overline{B}; \Pi \vdash \mathbf{lam} \, x.d : P_1 \to P_2 \ll \mu} \text{ (pr-lam)}$$

$$\frac{\Sigma; \overline{B}; \Pi \vdash d_1 : P_1 \to P_2 \ll \mu \quad \Sigma; \overline{B}; \Pi \vdash d_2 : P_1 \ll \mu}{\Sigma; \overline{B}; \Pi \vdash \mathbf{app}(d_1, d_2) : P_2 \ll \mu} \text{ (pr-app)}$$

$$\frac{\Sigma, \vec{a} : \vec{\sigma}; \overline{B}; \Pi, f : \forall \vec{a} : \vec{\sigma}.m \Rightarrow P \vdash d : P \ll \mu, f : s \quad m = (<, s)}{\Sigma; \overline{B}; \Pi \vdash \mathbf{fix} \, f[\vec{a}].m \Rightarrow d : \forall \vec{a} : \vec{\sigma}.P \ll \mu} \text{ (pr-fix)}$$

$$\frac{\Sigma; \overline{B}, B; \Pi \vdash d : P \ll \mu}{\Sigma; \overline{B}; \Pi \vdash \supset^+(d) : B \supset P \ll \mu} \text{ (pr-⊃-intro)}$$

$$\frac{\Sigma; \overline{B}; \Pi \vdash d : B \supset P \ll \mu \quad \Sigma; \overline{B} \models B}{\Sigma; \overline{B}; \Pi \vdash \supset^-(d) : P \ll \mu} \text{ (pr-⊃-elim)}$$

$$\frac{\Sigma, \vec{a} : \vec{\sigma}; \overline{B}; \Pi \vdash d : P \ll \mu}{\Sigma; \overline{B}; \Pi \vdash \mathbf{lam} \, \vec{a}.d : \forall \vec{a} : \vec{\sigma}.P \ll \mu} \text{ (pr-∀-intro)}$$

$$\frac{\Sigma, a : \sigma; \overline{B}; \Pi \vdash d : \forall \vec{a} : \vec{\sigma}.P \ll \mu \quad \Sigma \vdash \vec{s} : \vec{\sigma}}{\Sigma; \overline{B}; \Pi \vdash \mathbf{app}(d, \vec{s}) : P[\vec{a} \mapsto \vec{s}] \ll \mu} \text{ (pr-∀-elim)}$$

$$\frac{\Sigma; \overline{B} \models B \quad \Sigma; \overline{B}; \Pi \vdash d : P \ll \mu}{\Sigma; \overline{B}; \Pi \vdash \wedge(d) : B \wedge P \ll \mu} \text{ (pr-∧-intro)}$$

$$\frac{\Sigma; \overline{B}; \Pi \vdash d_1 : B \wedge P_1 \ll \mu \quad \Sigma; \overline{B}, B; \Pi, x : P_1 \vdash d_2 : P_2 \ll \mu}{\Sigma; \overline{B}; \Pi \vdash \mathbf{let} \ \wedge(x) = d_1 \ \mathbf{in} \ d_2 : P_2 \ll \mu} \text{ (pr-∧-elim)}$$

$$\frac{\Sigma \vdash s : \sigma \quad \Sigma; \overline{B}; \Pi \vdash d : P[a \mapsto s] \ll \mu}{\Sigma; \overline{B}; \Pi \vdash \langle s, d \rangle : \exists a : \sigma.P \ll \mu} \text{ (pr-∃-intro)}$$

$$\frac{\Sigma; \overline{B}; \Pi \vdash d_1 : \exists a : \sigma.P_1 \ll \mu \quad \Sigma, a : \sigma; \overline{B}; \Pi, x : P_1 \vdash d_2 : P_2 \ll \mu}{\Sigma; \overline{B}; \Pi \vdash \mathbf{let} \ \langle a, x \rangle = d_1 \ \mathbf{in} \ d_2 : P_2 \ll \mu} \text{ (pr-∃-elim)}$$

$$\frac{\Sigma \vdash s : bool \quad \Sigma; \overline{B}, s; \Pi \vdash d_1 : P \ll \mu \quad \Sigma; \overline{B}, \neg s; \Pi \vdash d_2 : P \ll \mu}{\Sigma; \overline{B}; \Pi \vdash \mathbf{sif}(s, d_1, d_2) : P \ll \mu} \text{ (pr-sta-if)}$$

Figure 6. The rules for assigning props to dynamic terms

2. If $\overline{B} = \overline{B}', \overline{B}''$ and $\Sigma; \overline{B}' \models \overline{B}''$ holds, then $\Sigma; \overline{B}'; \Pi \vdash d : P \ll \mu$ is derivable.

3. If $\Pi = \Pi', x : P'$ and $\Sigma; \overline{B}; \Pi' \vdash d' : P' \ll \mu$ holds, then $\Sigma; \overline{B}; \Pi' \vdash d[x \mapsto d'] : P \ll \mu$ is derivable.

4. If $\Pi = \Pi', f : \forall \Sigma_0.m \Rightarrow P'$ and $\mu = \mu', f : m$ and $\Sigma; \overline{B}; \Pi' \vdash d' : \forall \Sigma_0.P' \ll \mu'$ holds, then $\Sigma; \overline{B}; \Pi' \vdash d[x \mapsto d'] : P \ll \mu'$ is derivable.

Proof   By standard structural induction.   ■

We now introduce evaluation contexts as follows for assigning dynamic semantics to dynamic terms:

eval. ctx. E ::=
   $[] \mid dc[\vec{s}](v_1, \ldots, v_{i-1}, E, d_i, \ldots, d_n) \mid \mathbf{app}(E, d) \mid \mathbf{app}(v, E) \mid$
   $\supset^-(E) \mid \mathbf{app}(E, \vec{s}) \mid \mathbf{let} \wedge (x) = E \mathbf{\ in\ } d \mid \mathbf{let} \langle a, x \rangle = E \mathbf{\ in\ } d$

DEFINITION 4.   We define redexes and their reducts as follows.

1. $dcf(v_1, \ldots, v_n)$ is a redex if it is defined, and its reduct is its defined value.

2. $\mathbf{app}(\mathbf{lam}\ x.d, v)$ is a redex, and its reduct is $d[x \mapsto v]$.

3. $\mathbf{fix}\ f[\vec{a}].m \Rightarrow d$ is a redex, and its reduct is:

$$\mathbf{lam}\ \vec{a}.d[f \mapsto \mathbf{fix}\ f[\vec{a}].m \Rightarrow d]$$

4. $\supset^-(\supset^+(d))$ is a redex and its reduct is d.

5. $\mathbf{app}(\mathbf{lam}\ \vec{a}.d, \vec{s})$ is a redex, and its reduct is $d[\vec{a} \mapsto \vec{s}]$.

6. $\mathbf{let}\ \wedge (x) = \wedge(v) \mathbf{\ in\ } d$ is a redex and its reduct is $d[x \mapsto v]$.

7. $\mathbf{let}\ \langle a, x \rangle = \langle s, v \rangle \mathbf{\ in\ } d$ is a redex and its reduct is $d[a \mapsto s][x \mapsto v]$.

8. $\mathbf{sif}(tt, d_1, d_2)$ is a redex, and its reduct is d_1.

9. $\mathbf{sif}(ff, d_1, d_2)$ is a redex, and its reduct is d_2.

Given $d_1 = E[d]$ and $d_2 = E[d']$, where d is a redex and d' is the reduct of d, we write $d_1 \rightarrow d_2$ and say that d_1 reduces to d_2 in one step. Let $\rightarrow^*$ be the reflexive and transitive closure of $\rightarrow$.

THEOREM 5 (Subject Reduction). *Assume that $\Sigma; \overline{B}; \Pi \vdash d : P \ll \mu$ is derivable and $d \rightarrow d'$ holds. Then $\Sigma; \overline{B}; \Pi \vdash d' : P \ll \mu$ is also derivable.*

Proof By structural induction on the derivation of $\Sigma; \overline{B}; \Pi \vdash d : P$. ∎

THEOREM 6 (Progress). *Assume that $\emptyset; \emptyset; \emptyset \vdash d : P$ is derivable. Then d is either a value or $d \rightarrow d'$ holds for some d'.*

Proof By structural induction on the derivation of $\emptyset; \emptyset; \emptyset \vdash d : P$. ∎

We next establish that every closed dynamic term d can be reduced to a value v if d can be assigned a prop in ATS/LF. The proof technique we employ is based on the notion of reducibility [30]. Given a dynamic term d, we write $d \downarrow$ to mean that there is *no* infinite reduction sequence from d: $d = d_0 \rightarrow d_1 \rightarrow d_2 \rightarrow \dots$.

DEFINITION 7 (Reducibility). Suppose that d is a closed dynamic term of prop P, that is, $\emptyset; \emptyset; \emptyset \vdash d : P$ is derivable. We define that d is reducible of prop P by induction on the complexity of P, namely, the number of prop constructors $\rightarrow, \supset, \forall, \wedge$ and $\exists$ in P.

1. P is a base prop. Then d is reducible of prop P if $d \downarrow$ holds.

2. $P = P_1 \rightarrow P_2$. Then d is reducible of prop P if $d \downarrow$ holds and $d_0[x \mapsto v]$ is reducible of prop P_2 for any value v reducible of prop P_1 whenever $d \rightarrow^* \mathbf{lam}\, x.d_0$ holds for some d_0.

3. $P = B \supset P_0$. Then d is reducible of prop P if $d \downarrow$ holds and $\models B$ implies that d_0 is reducible of prop P_0 whenever $d \rightarrow^* \supset^+(d_0)$ holds for some d_0.

4. $P = B \wedge P_0$. Then d is reducible of prop P if $d \downarrow$ holds and v is reducible of prop P_0 whenever $d \rightarrow^* \wedge(v)$ holds.

5. $P = \forall \vec{a} : \vec{\sigma}.\, P_0$. Then d is reducible of prop P if $d \downarrow$ holds and $d_0[\vec{a} \mapsto \vec{s}]$ is reducible of prop $P_0[\vec{a} \mapsto \vec{s}]$ for any $\vec{s}$ of sorts $\vec{\sigma}$ whenever $d \rightarrow^* \mathbf{lam}\, \vec{a}.d_0$ holds for some d_0.

6. $P = \exists a : \sigma.P_0$. Then d is reducible of prop P if $d \downarrow$ holds and v is reducible of prop $P_0[a \mapsto s]$ whenever $d \rightarrow^* \langle s, v \rangle$ holds for some s and v.

For handling the fixed-point construct, we also introduce a notion of labeled reducibility as follows.

DEFINITION 8 (Labeled Reducibility). Assume that $\mathbf{fix}\, f[\vec{a}].m \Rightarrow d$ is a closed dynamic term of prop $\forall \vec{a} : \vec{\sigma}.P$. Given a label s_1, d is s_1-reducible

of prop $\forall \vec{a} : \vec{\sigma}.P$ if $\mathbf{app}(\mathbf{fix}\, f[\vec{a}].m \Rightarrow d, \vec{s})$ is reducible of prop $P[\vec{a} \mapsto \vec{s}]$ for each $\vec{s}$ satisfying $s_0[\vec{a} \mapsto \vec{s}] < s_1$, where $m = (<, s_0)$.

PROPOSITION 9. *Assume that $\emptyset; \emptyset \vdash P \leq_{pr} P'$ is derivable and d is reducible of prop P. Then d is also reducible of prop P'.*

Proof By induction on the height of the derivation of $\emptyset; \emptyset \vdash P \leq_{pr} P'$. ∎

The following lemma is the key to establishing Theorem 11, the main theoretical result in this paper:

LEMMA 10 (Main). *Assume that $\Sigma; \overline{B}; \Pi \vdash d : P \ll \mu$ is derivable. Given Θ and θ such that $\Theta : \Sigma$ and $\theta : \Pi[\Theta]$, if we have*

1. *$\models \overline{B}[\Theta]$ holds, and*

2. *for each $x \in \mathbf{dom}(\Pi)$, $\theta(x)$ is reducible of prop $\Pi(x)$, and*

3. *for each $f \in \mathbf{dom}(\Pi)$, $\theta(f)$ is $\mu(f)[\Theta]$-reducible of prop $\forall \vec{a} : \vec{\sigma}.P$, where $\Pi(f) = \forall \vec{a} : \vec{\sigma}.m \Rightarrow P$,*

then $d[\Theta][\theta]$ is reducible of prop $P[\Theta]$.

Proof By structural induction on the derivation of $\Sigma; \overline{B}; \Pi \vdash d : P \ll \mu$. Proposition 9 is needed for handling the rule **(pr-sub)**. Please see [34] for details in a closely related proof.[3] ∎

THEOREM 11 (Totality). *Assume that $\emptyset; \emptyset; \emptyset \vdash d : P$ is derivable. Then $d \rightarrow^* v$ holds for some value v.*

Proof By Lemma 10, we have that d is reducible of prop P. Then by the definition of reducibility, $d \downarrow$ holds, and by Theorem 6, we have $d \rightarrow^* v$ for some value v. ∎

4 Dataprops and Pattern Matching

We find that dataprops (similar to datatypes) and pattern matching are indispensable in practice. The following is some additional syntax we need for introducing these features:

patterns	p ::=	$x \mid dcc[\vec{a}](p_1, \ldots, p_n)$
dynamic terms	d ::=	$\ldots \mid \mathbf{case}\, d_0\, \mathbf{of}\, p_1 \Rightarrow d_1 \mid \cdots \mid p_n \Rightarrow d_n$
eval. ctx.	E ::=	$\ldots \mid \mathbf{case}\, E\, \mathbf{of}\, p_1 \Rightarrow d_1 \mid \cdots \mid p_n \Rightarrow d_n$

As usual, we require that any variable, static or dynamic, occur at most once in a pattern. Given a value v and a pattern p, we use a judgment of

---

[3]The proof of Lemma 3.9.

$$\overline{\Sigma \vdash x \Downarrow P \Rightarrow \emptyset; \emptyset; \emptyset, x : P} \ \textbf{(pat-var)}$$

$$\vdash dcc : \forall \vec{a} : \vec{\sigma}. \overline{B}_0 \supset ((P_1, \ldots, P_n) \Rightarrow \delta(\vec{s}_0))$$
$$\Sigma, \vec{a} : \vec{\sigma} \vdash p_i \Downarrow P_i \Rightarrow \Sigma_i; \overline{B}_i; \Pi_i \ \text{ for } 1 \leq i \leq n$$
$$\frac{\Sigma' = \Sigma_1, \ldots, \Sigma_n \quad \overline{B}' = \overline{B}_1, \ldots, \overline{B}_n \quad \Pi' = \Pi_1, \ldots, \Pi_n}{\Sigma \vdash dcc[\vec{a}](p_1, \ldots, p_n) \Downarrow \delta(\vec{s}) \Rightarrow \vec{a} : \vec{\sigma}, \Sigma'; \overline{B}_0, \vec{s}_0 = \vec{s}, \overline{B}'; \Pi'} \ \textbf{(pat-dcc)}$$

$$\frac{\Sigma \vdash p \Downarrow P_1 \Rightarrow \Sigma'; \overline{B}'; \Pi' \quad \Sigma, \Sigma'; \overline{B}, \overline{B}'; \Pi, \Pi' \vdash d : P_2}{\Sigma; \overline{B}; \Pi \vdash p \Rightarrow d \Downarrow P_1 \Rightarrow P_2} \ \textbf{(pr-clause)}$$

$$\frac{\Sigma; \overline{B}; \Pi \vdash d_0 : P_1 \quad \Sigma; \overline{B}; \Pi \vdash p_i \Downarrow d_i : P_1 \Rightarrow P_2 \ \text{ for } 1 \leq i \leq n}{\Sigma; \overline{B}; \Pi \vdash (\textbf{case } d_0 \textbf{ of } p_1 \Rightarrow d_1 \mid \cdots \mid p_n \Rightarrow d_n) : P_2} \ \textbf{(pr-case)}$$

Figure 7. The proping rules for pattern matching

the form $v \Downarrow p \Rightarrow (\Theta; \theta)$ to indicate $v = p[\Theta][\theta]$. The rules for deriving such judgments are given as follows,

$$\overline{v \Downarrow x \Rightarrow ([]; [x \mapsto v])} \ \textbf{(vp-var)}$$

$$v_i \Downarrow p_i \Rightarrow (\Theta_i; \theta_i) \ \text{ for } 1 \leq i \leq n$$
$$\frac{\Theta = [\vec{a} \mapsto \vec{s}] \cup \Theta_1 \cup \ldots \cup \Theta_n \quad \theta = \theta_1 \cup \ldots \cup \theta_n}{dcc[\vec{s}](v_1, \ldots, v_n) \Downarrow dcc[\vec{a}](p_1, \ldots, p_n) \Rightarrow (\Theta; \theta)} \ \textbf{(vp-dcc)}$$

and we say that v matches p if $v \Downarrow p \Rightarrow (\Theta; \theta)$ is derivable for some Θ and θ. Note that in the rule **(vp-dcc)**, the unions $\Theta_1 \cup \ldots \cup \Theta_n$ and $\theta_1 \cup \ldots \cup \theta_n$ are well-defined since any variable can occur at most once in a pattern.

A dynamic term of the form **case** v **of** $p_1 \Rightarrow d_1 \mid \cdots \mid p_n \Rightarrow d_n$ is a redex if $v \Downarrow p_i \Rightarrow \theta$ holds for some $1 \leq i \leq n$, and its reduction is $d_i[\theta]$. Note that reducing such a redex may involve nondeterminism as v may match several patterns p_i.

The proping rules for pattern matching is given in Figure 7. The meaning of a judgment of the form $\Sigma \vdash p \Downarrow s \Rightarrow \Sigma'; \overline{B}'; \Pi'$ is formally captured by the following lemma.

LEMMA 12. *Assume* $\emptyset; \emptyset; \emptyset \vdash v : P$, $\emptyset \vdash p \Downarrow P \vdash \Sigma; \overline{B}; \Pi$ *and* $v \Downarrow p \Rightarrow (\Theta; \theta)$. *Then we have* $\Theta : \Sigma, \models \overline{B}[\Theta]$ *and* $\theta : \Pi[\Theta]$.

Proof By structural induction.   ■

As an example, the judgment below is derivable,

$$m' : int, n' : int, p' : int \vdash MULind(x) \Downarrow \mathbf{MUL}(m', n', p') \Rightarrow \Sigma; \overline{B}; \Pi$$

where $MULind$ is assumed to have the following c-prop:

$$\forall m : int.\forall n : int.\forall p : int.(\mathbf{MUL}(m, n, p)) \Rightarrow \mathbf{MUL}(m + 1, n, p + n)$$

and $\Sigma = (m : int, n : int, p : int)$, $\overline{B} = (m + 1 = m', n = n', p + n = p')$ and $\Pi = (x : \mathbf{MUL}(m, n, p))$.

In order to guarantee that a closed well-typed dynamic term of the following form:

$$\mathbf{case}\ v\ \mathbf{of}\ p_1 \Rightarrow d_1 \mid \cdots \mid p_n \Rightarrow d_n$$

is always a redex, we need to verify the exhaustiveness of pattern matching in the rule **(pr-case)**. We refer the reader to [33, 35] for more details on this issue. From now on, we assume the exhaustiveness of pattern matching is properly verified when the rule **(pr-case)** is applied.

In the presence of pattern matching, nonterminating programs can be readily constructed if we impose no restrictions on dataprops. We thus need to propose some restrictions that can be imposed to disallow nonterminating programs.

DEFINITION 13. An occurrence δ in a prop P is ground if $P = \delta(\vec{s})$, or $P = \Sigma a : \sigma.P_0$ and the occurrence is ground in P_0. An occurrence δ in $\vec{P}$ is ground if the occurrence is ground in some P in $\vec{P}$.

Let us fix a recursive prop constructor δ_0 that takes static terms $\vec{s}$ of sorts $\vec{\sigma}_0$ to form a prop $\delta_0(\vec{s})$. Also, we assume that dcc_k $(k = 1, \ldots, m)$ of c-props $\forall \vec{a}_k : \vec{\sigma}_k. (\vec{P}_k) \Rightarrow \delta_0(\vec{s}_k)$ are associated with δ_0. Then Theorem 11 still holds after δ_0 and dcc_k $(k = 1, \ldots, m)$ are added into ATS/LF if

1. all occurrences of δ_0 in $\vec{P}_k$ are ground, or

2. there is a metric $m = (<, s_0)$ such that $\vec{a} : \vec{\sigma}_0 \vdash s_0 : \sigma$ is derivable and for each occurrence of $\delta(\vec{s})$ in $\vec{P}_k$, $s_0[\vec{a} \mapsto \vec{s}] < s_0[\vec{a} \mapsto \vec{s}_k]$ holds.

The reason for this claim is that it is possible to define the notion of reducibility of prop $\delta(\vec{s})$ for all $\vec{s} : \vec{\sigma}_0$ if either of these two criteria is satisfied. The detailed justification for the first criterion can be found in [34]. As for the second criterion, its justification is rather similar to that of the first one. For instance, the first criterion is satisfied if δ_0 is **MUL**. In practice, it seems uncommon to encounter a need for the second criterion. A genuine case that requires the second criterion occurs in an encoding of the reducibility predicate [30] for the simply-typed lambda-calculus [13].

```
// [lemma1_commute] proves: m * 0 = p implies p = 0
prfun lemma1_commute {m:nat} {p:int} .<m>.
   (pf: MUL (m, 0, p)): [p == 0] void =
  case pf of
  | MULbas () => ()
  | MULind (pf) => let val _ = lemma1_commute (pf) in () end

// [lemma2_commute] proves: m * n = p implies m * (n-1) = p-m
prfun lemma2_commute {m,n:nat} {p:int} .<m>.
   (pf: MUL (m, n, p)): MUL (m, n-1, p-m) =
  case pf of
  | MULbas () => MULbas ()
  | MULind pf => MULind (lemma2_commute pf)

// [lemma3_commute] proves: m * n = p implies m * (-n) = -p
prfun lemma3_commute {m,n:nat} {p:int} .<m>.
   (pf: MUL (m, n, p)): MUL (m, ~n, ~p) =
  case pf of
  | MULbas () => MULbas ()
  | MULind pf => MULind (lemma3_commute pf)

// [theorem_commute] proves:
//    m * n = p1 and n * m = p2 implies p1 = p2
prfun theorem_commute {m,n:nat} {p1,p2:int} .<m>.
  (pf1: MUL (m, n, p1), pf2: MUL (n, m, p2)): [p1==p2] void =
  case pf1 of
  | MULbas () => let
      prval () = lemma1_commute pf2
    in
      ()
    end
  | MULind pf1 => let
      prval pf2 = lemma2_commute pf2
    in
      theorem_commute (pf1, pf2)
    end
  | MULneg pf1 => let
      prval pf2 = lemma3_commute pf2
    in
      theorem_commute (pf1, pf2)
    end
```

Figure 8. A proof of the commutativity of multiplication

5 ATS/LF as a (meta) logical framework

We use some short examples in this section to illustrate how deduction systems can be encoded in ATS/LF.

5.1 Arithmetic

In Figure 8, we present some code for proving that multiplication on natural numbers is commutative. We first establish three lemmas that prove the following:

1. $\forall m : nat.\forall p : int.\ m \times 0 = p \supset p = 0$, and

2. $\forall m : nat.\forall n : nat.\forall p : int.\ m \times n = p \supset m \times (n-1) = p - m$, and

3. $\forall m : nat.\forall n : nat.\forall p : int.\ m \times n = p \supset m \times (-n) = -p.$

where the induction is on m. Then a theorem is proven that states:

$$\forall m : nat.\forall n : nat.\forall p_1 : int.\forall p_2 : int.\ m \times n = p_1 \wedge m \times n = p_2 \supset p_1 = p_2$$

In other words, the commutativity of multiplication on natural numbers is established. If we go a bit further, we can readily use the commutativity of multiplication to construct the following proof function and thus establish the irrationality of the square root of 2:

```
// for proving that the square root of 2 is irrational
prfun lemma_irrational {p,q:nat} {x:int}
  (pf1: MUL (p, p, x), pf2: MUL (q, q, 2*x)): [x == 0] void =
  ...
```

5.2 Sequent Calculus

We present some code in Figure 9 that illustrates an approach to encoding a fragment of intuitionistic sequent calculus in ATS/LF. This approach is largely adopted from [7]. The syntax for this fragment is given as follows:

$$
\begin{array}{llll}
\text{formulas} & \alpha & ::= & \dots \mid \alpha_1 \supset \alpha_2 \\
\text{formula sequences} & \Gamma & ::= & \emptyset \mid \Gamma, \alpha
\end{array}
$$

For brevity, we only support the implication logic connective here. We first declare a sort *form* for representing (propositional) formulas and another sort *forms* for representing sequences of formulas. Given the representations being first-order, we skip the issue of representation adequacy as it is evident. We then declare a dataprop **IN** such that given a formula A and a sequence G of formulas, **IN**(A, G) is a prop that indicates A occurring in G if a closed proof value of prop **IN**(A, G) can be constructed.

```
datasort form = imp of (form, form)
datasort forms = none | more of (forms, form)

datatype IN (form, forms) =
  | {G:forms; A:form} INone (A, more (G, A))
  | {G:forms; A1,A2:form} INshi (A1, more (G, A2)) of IN (A1, G)

datatype DER (forms, form, int) =
  | {G:forms; A:form} axiom (G, A, 0) of IN (A, G)
  | {G:forms; A1,A2,A3:form; n1,n2:nat}
    impl (G, A3, n1+n2+1) of
      (IN (imp (A1,A2),G), DER (G,A1,n1), DER (more (G,A2),A3,n2))
  | {G:forms; A1,A2:form; n:nat}
    impr (G, imp (A1, A2), n+1) of DER (more (G, A1), A2, n)

// [sup (G1, G2)] means that G1 contains G2
propdef SUP (G1:forms, G2:forms) = {A:form} IN (A, G2) -> IN (A, G1)

prfun shiSUP {G1,G2:forms; A:form} .<>.
   (f: SUP (G1, G2)): SUP (more (G1, A), more (G2, A)) =
  lam i => case i of INone () => INone () | INshi i => INshi (f i)

prfun supDER {G1,G2:forms; A:form; n:nat} .<n>.
   (f: SUP (G1, G2), d: DER (G2, A, n)): DER (G1, A, n) =
  case d of
  | axiom i => axiom (f i)
  | impl (i, d1, d2) =>
    impl (f i, supDER (f, d1), supDER (shiSUP f, d2))
  | impr (d) => impr (supDER (shiSUP f, d))

prfun cutElim {G:forms; A1,A2:form; n1,n2:nat} .<A1, n1, n2>.
   (d1: DER (G, A1), d2: DER (more (G, A1), A2)): DER (G, A2) =
  ...
```

Figure 9. An encoding of sequent calculus in ATS/LF

   To represent derivations in sequent calculus, we declare a dataprop **DER**; given G, A and n, a derivation for $\Gamma \vdash \alpha$ of size n can be constructed if there is a proof value of prop **DER**(G, A, n), where we assume G and A represent Γ and α, respectively. In particular, the three constructors *axiom*, *impl* and *impr* represent the following three rules **(axiom)**, **(impl)** and **(impr)**,

respectively:

$$\frac{\Gamma \vdash \alpha}{\alpha \in \Gamma} \text{ (axiom)}$$

$$\frac{\alpha_1 \supset \alpha_2 \in \Gamma \quad \Gamma \vdash \alpha_1 \quad \Gamma, \alpha_2 \vdash \alpha_3}{\Gamma \vdash \alpha_3} \text{ (impl)}$$

$$\frac{\Gamma, \alpha_1 \vdash \alpha_2}{\Gamma \vdash \alpha_1 \supset \alpha_2} \text{ (impr)}$$

As an example, we implement a function *supDER* of the following prop:

$$\forall G_1 : forms.\forall G_2 : forms.\forall A : form.\forall n : nat.$$
$$(\mathbf{SUP}(G_1, G_2), \mathbf{DER}(G_2, A, n)) \rightarrow \mathbf{DER}(G_1, A, n)$$

where $\mathbf{SUP}(G_1, G_2)$ stands for $\forall A : form.\ \mathbf{IN}(A, G_2) \rightarrow \mathbf{IN}(A, G_1)$. There-
fore, *supDER* encodes a proof of the following statement: *If* Γ_1 *contains* Γ_2
and there is a derivation for $\Gamma_2 \vdash \alpha$ *of size n, then there is also a derivation*
for $\Gamma_1 \vdash \alpha$ *of size n.* With this, the admissibility of following structural
rules can be readily established:

$$\frac{\Gamma \vdash \alpha_2}{\Gamma, \alpha_1 \vdash \alpha_2} \text{ (weak.)} \quad \frac{\Gamma, \alpha_1, \alpha_1 \vdash \alpha_2}{\Gamma, \alpha_1 \vdash \alpha_2} \text{ (contr.)} \quad \frac{\Gamma, \alpha_1, \alpha_2 \vdash \alpha_3}{\Gamma, \alpha_2, \alpha_1 \vdash \alpha_3} \text{ (exch.)}$$

Clearly, the availability of higher-order functions in ATS/LF can be easily
appreciated in this example. It is straightforward to construct a function
cutElim of the following prop:

$$\forall G : forms.\forall A_1 : form.\forall A_2 : form.\forall n_1 : nat.\forall n_2 : nat.$$
$$(\mathbf{DER}(G, A_1, n_1), \mathbf{DER}(G, A_2, n_2)) \rightarrow \exists n : nat.\ \mathbf{DER}(G, A_1, n)$$

That is, *cutElim* encodes a proof of the cut elimination theorem for intuition-
istic sequent calculus. In particular, the metric for verifying the termination
of *cutElim* is the triple $\langle A_1, n_1, n_2 \rangle$, lexicographically ordered (the ordering
on formulas is the standard structural ordering). The interested reader may
find further details in [7].

5.3   Lambda-Calculus

We now present an interesting example that involves the use of higher-
order abstract syntax [8, 25, 26]. In Figure 10, we declare a sort *tm* for
representing pure λ-terms in the statics of ATS/LF. For instance, the
lambda-term $\lambda x.\lambda y.y(x)$ is represented as:

$$TMlam(\lambda a_1 : tm.\,TMlam(\lambda a_2 : tm.\,TMapp(a_2, a_1)))$$

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

dataprop EVAL (tm, tm) =
  | {f:tm -> tm} EVALlam (TMlam f, TMlam f)
  | {t1,t2,t3:tm; f: tm -> tm}
    EVALapp (TMapp (t1, t2), t3) of
      (EVAL (t1, TMlam f), EVAL (f t2, t3))

datasort tp = TPzero | TPfun of (tp, tp)

// This is a representation *not* to be used
dataprop TPDER0 (tm, tp) =
  | {f: tm -> tm; T1,T2:tp}
    TPDER0lam (TMlam f, TPfun (T1, T2)) of
      {x:tm} TPDER0 (x, T1) -> TPDER0 (f x, T2)
  | {t1,t2:tm; T1,T2:tp}
    TPDER0app (TMapp (t1, t2), T2) of
      (TPDER0 (t1, TPfun (T1, T2)), TPDER0 (t2, T1))

datasort ctx = CTXemp | CTXmore of (ctx, tm, tp)

dataprop IN (tm, tp, ctx) =
  | {G:ctx; t:tm; T:tp} INone (t, T, CTXmore (G, t, T))
  | {G:ctx; t,t':tm; T,T':tp}
    INshi (t, T, CTXmore (G, t', T')) of IN (t, T, G)

// This is the representation to be actually used
dataprop TPDER (ctx, tm, tp, int) =
  | {G:ctx; t:tm; T:tp} TPDERhyp (G, t, T, 0) of IN (t, T, G)
  | {G:ctx; f:tm -> tm; T1,T2:tp; n:nat}
    TPDERlam (G, TMlam f, TPfun (T1, T2), n+1) of
      {x:tm} TPDER (CTXmore (G, x, T1), f x, T2, n)
  | {G:ctx; t1,t2:tm; T1,T2:tp; n1,n2:nat}
    TPDERapp (G, TMapp (t1, t2), T2, n1+n2+1) of
      (TPDER (G, t1, TPfun (T1, T2), n1), TPDER (G, t2, T1, n2))

prfun SubstitutionLemma {G:ctx} {t1,t2:tm} {T1,T2:tp}
  (d1: TPDER (G, t1, T1), d2: TPDER (CTXmore (G, t1, T1), t2, T2))
  : TPDER (G, t2, T2) = ...

prfun TypePreservation {t1,t2:tm} {T:tp}
  (eval (t1, t2), d: TPDER (CTXemp, t1, T))
  : TPDER (CTXemp, t2, T) = ...
```

Figure 10. An encoding of lambda-calculus in *ATS/LF*

The essence in this representation is that a variable at the object level (the lambda-calculus) is represented by a variable at the meta level (the statics of ATS/LF). A particularly appealing feature of this representation is that a substitution function at object-level can often be readily supported by a (built-in) substitution function at meta level. For instance, if $TMlam(f)$ and t represent $\lambda x.M_1$ and M_2, then $f(t)$ represents $M_1[x \mapsto M_2]$. Of course, the adequacy of such a representation needs to be formally justified, which can be found in [24]. As an example, we declare a prop constructor **EVAL** that takes two static terms s_1 and s_2 of the sort tm to form a prop **EVAL**(s_1, s_2); if a closed value of prop **EVAL**(s_1, s_2) can be constructed, then the weak head normal form (WHNF) of the lambda-term represented by s_1 is the lambda-term represented by s_2.

Suppose that we now want to assign simple types to lambda-terms. We declare a datasort tp in Figure 10 for representing simple types; $TPzero$ represent a base type and $TPfun(\cdot, \cdot)$ represents a function type.

In order to represent typing derivations, we declare a prop constructor **TPDER**$_0$ that takes two static terms t and T of sorts tm and tp, respectively, to form a prop **TPDER**$_0(t, T)$; a proof value of prop **TPDER**$_0(t, T)$ is *intended* to represent a typing derivation that assigns the type represented by T to the lambda-term represented by t. Note that this is a higher-order representation as $TPDER0lam$ is assigned the following c-prop:

$$\forall f : tm \to tm.\forall T_1 : tp.\forall T_2 : tp.$$
$$(\forall t : tm. \ \textbf{TPDER}_0(t, T_1) \to \textbf{TPDER}_0(f(t), T_2)) \Rightarrow$$
$$\textbf{TPDER}_0(\mathit{TMlam}(t), \mathit{TPfun}(T_1, T_2))$$

There are some rather serious problems with this representation. First and foremost, the adequacy of this representation is difficult to establish (even if it holds). Second, neither of the two criteria in Section 4 is satisfied, and thus we need additional techniques for proving the totality of functions that involve the use of **TPDER**$_0$.

To avoid these difficult issues, we introduce a first-order representation for typing derivations that assign simple types to lambda-terms. We first declare a sort ctx for representing contexts; $CTXemp$ represents the empty context and $CTXmore(G, t, T)$ represents the context that extends G with a declaration for a typing derivation that assigns the type represented by T to the lambda-term represented by t. We then declare a prop constructor **TPDER** that takes four static terms G, t, T and n of sorts ctx, tm, tp and int, respectively, to form a prop **TPDER**(G, t, T, n); a value of prop **TPDER**(G, t, T, n) represents a typing derivation of size n that assigns the type represented by T to the lambda-term represented by t, where G indicates that the derivation may contain indeterminates (of the form

TPDERhyp(···)) standing for typing derivations declared in G. With this representation for typing derivations, the substitution lemma and the type preservation theorem for simply typed lambda-calculus can be readily encoded in ATS/LF as two functions of the following props, respectively:

$$\forall G : ctx.\forall t_1 : tm.\forall t_2 : tm.\forall T_1 : tp.\forall T_2 : tp.$$
$$(\mathbf{TPDER}^*(G, t_1, T_1), \mathbf{TPDER}^*(CTXmore(G, t_1, T_1), t_2, T_2)) \rightarrow$$
$$\mathbf{TPDER}^*(G, t_2, T_2)$$

$$\forall t_1 : tm.\forall t_2 : tm.\forall T : tp.$$
$$(\mathbf{EVAL}(t_1, t_2), \mathbf{TPDER}^*(CTXemp, t_1, T)) \rightarrow$$
$$\mathbf{TPDER}^*(CTXemp, t_2, T)$$

where $\mathbf{TPDER}^*(G, t, T) = \exists n : nat.\ \mathbf{TPDER}(G, t, T, n)$. The interested reader should have no difficulty in filling out the details.

6   Conclusion

We have presented a type system ATS/LF rooted in the framework *Applied Type System* [36, 31]. In ATS/LF, every well-typed program is guaranteed to be total. When constructing a recursive function in ATS/LF, the programmer is required to provide a metric for verifying that the recursive function is terminating. This is essentially the approach to program termination advocated in [34]. We, however, have given an account of this approach in a more general setting. While the primary motivation for developing ATS/LF is to support a programming paradigm that allows programs to be combined with proofs, we argue that ATS/LF can also be used as a logical framework for encoding deduction systems and their properties. This application is not attempted in [34]. As of now, there is no support for parametric polymorphism in ATS/LF. We expect to incorporate this feature into ATS/LF in the near future, and we consider such an incorporation to be largely straightforward.

BIBLIOGRAPHY

[1] Peter Aczel. An Introduction to Inductive Definition. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North Holland Publishing Company, 1977.

[2] Peter B Andrews. General Models, Descriptions and Choice in Type Theory. *Journal of Symbolic Logic*, 37:385–394, 1972.

[3] Peter B. Andrews. *An Introduction to Mathematical Logic: To Truth through Proof*. Academic Press, Inc., Orlando, Florida, 1986.

[4] Hendrik Pieter Barendregt. Lambda Calculi with Types. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 117–441. Clarendon Press, Oxford, 1992.

[5] Ana Bove. *General Recursion in Type Theory*. Ph. D. Dissertation, Chalmers University of Technology, November 2002.

[6] 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.

[7] Chiyan Chen, Dengping Zhu, and Hongwei Xi. 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.

[8] Alonzo Church. A formulation of the simple type theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[9] Robert L. Constable et al. *Implementing Mathematics with the NuPrl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[10] Robert L. Constable and Scott Fraser Smith. Partial Objects In Constructive Type Theory. In *Proceedings of Symposium on Logic in Computer Science*, pages 183–193, Ithaca, New York, June 1987.

[11] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3):95–120, February–March 1988.

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

[13] Kevin Donnelly and Hongwei Xi. A Formalization of Strong Normalization for Simply Typed Lambda-Calculus and System F. In *Proceedings of Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, pages 109–125. ENTCS 174(5), 2006.

[14] 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.

[15] Susumu Hayashi and Hiroshi Nakano. *PX: A Computational Logic*. The MIT Press, 1988.

[16] Leon Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81–91, 1950.

[17] Furio Honsell, Ian A. Mason, Scott Smith, and Carolyn Talcott. A Variable Typed Logic of Effects. *Information and Computation*, 119(1):55–90, 15 May 1995.

[18] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, Italy, 1984.

[19] N.P. Mendler. Recursive Types and Type Constraints in Second-Order Lambda Calculus. In *Proceedings of Symposium on Logic in Computer Science*, pages 30–36, Ithaca, New York, June 1987. The Computer Society of the IEEE.

[20] Robin Milner, Mads Tofte, Robert W. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[21] Bengt Nordström. Terminating General Recursion. *BIT*, 28(3):605–619, October 1988.

[22] 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.

[23] Lawrence Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[24] Frank Pfenning. *Computation and Deduction*. Lecture Notes, 2002.

[25] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pages 199–208, Atlanta, Georgia, June 1988.

[26] Frank Pfenning and Peter Lee. A Language with Eval and Polymorphism. In *International Joint Conference on Theory and Practice in Software Development*, pages 345–359, Barcelona, Spain, March 1989. Springer-Verlag LNCS 352.

[27] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[28] Brigitte Pientka and Frank Pfenning. Termination and Reduction Checking in the Logical Framework. In *Proceedings of Workshop on Automation of Proofs by Mathematical Induction*, Pittsburgh, PA, June 2000.

[29] Carsten Schürmann et al. Delphin Project. Available at: `http://www.cs.yale.edu/~delphin`.

[30] W. W. Tait. Intensional Interpretations of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.

[31] Hongwei Xi. Applied Type System. Available at: `http://www.cs.bu.edu/~hwxi/academic/drafts/ATS.ps`.

[32] Hongwei Xi. Applied Type System. Available at:`http://www.ats-lang.org/`.

[33] Hongwei Xi. 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.

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

[35] Hongwei Xi. Dependently Typed Pattern Matching. *Journal of Universal Computer Science*, 9(8):851–872, 2003.

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

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

[38] 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, San Antonio, Texas, January 1999. ACM press.

[39] Dengping Zhu and Hongwei Xi. Safe Programming with Pointers through Stateful Views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, Long Beach, CA, January 2005. Springer-Verlag LNCS, 3350.