

ATS: A Language that Combines Programming with Theorem Proving

Sa Cui, Kevin Donnelly and Hongwei Xi

Computer Science Department
Boston University
{cuisa,kevind,hwxi}@cs.bu.edu

Abstract. ATS is a language with a highly expressive type system that supports a restricted form of dependent types in which programs are not allowed to appear in type expressions. The language is separated into two components: a proof language in which (inductive) proofs can be encoded as (total recursive) functions that are erased before execution, and a programming language for constructing programs to be evaluated. This separation enables a paradigm that combines programming with theorem proving. In this paper, we illustrate by example how this programming paradigm is supported in ATS.

1 Introduction

The framework *Pure Type System* (*PTS*) [1] offers a simple and general approach to designing and formalizing type systems. However, *PTS* makes it difficult, especially, in the presence of dependent types to accommodate many common realistic programming features, such as general recursion [7], recursive types [11], effects [10] (e.g., exceptions [9], references, input/output), etc. To address such limitations of *PTS*, the framework *Applied Type System* (*ATS*) [14] has been proposed to allow for designing and formalizing (advanced) type systems in support of practical programming. The key salient feature of *ATS* lies in a complete separation of the statics, in which types are formed and reasoned about, from the dynamics, in which programs are constructed and evaluated. With this separation, it is no longer possible for programs to occur in type expressions as is otherwise allowed in *PTS*.

Currently, *ATS*, a language with a highly expressive type system rooted in the framework *ATS*, is under active development. In *ATS*, a variety of programming paradigms are supported in a typeful manner, including functional programming, object-oriented programming [3], imperative programming with pointers [16] and modular programming. There is also a theorem proving component in *ATS* [4] that allows the programmer to encode (inductive) proofs as (total recursive) functions, supporting a paradigm that combines programming with theorem proving [5]. This is fundamentally different from the paradigm of extracting programs from proofs as is done in systems such as Coq [2] and NuPrl [6]. In *ATS*, proofs are completely erased before execution, while proofs in Coq, for example, are not. In addition, *ATS* allows the construction of programs involving

sorts	$\sigma ::= b \mid \sigma_1 \rightarrow \sigma_2$
static terms	$s ::= a \mid \lambda a : \sigma. s \mid s_1(s_2) \mid sc(s_1, \dots, s_n)$
sta. var. ctx.	$\Sigma ::= \emptyset \mid \Sigma, a : \sigma$
dynamic terms	$d ::= x \mid dc(d_1, \dots, d_n) \mid \mathbf{lam} x.d \mid \mathbf{fix} x.d \mid \mathbf{app}(d_1, d_2) \mid \lambda a : \sigma.d \mid d(s) \mid \dots$
values	$v ::= x \mid dcc(v_1, \dots, v_n) \mid \mathbf{lam} x.d \mid \dots$
dyn. var. ctx.	$\Delta ::= \emptyset \mid \Delta, x : T$

Fig. 1. Abstract syntax for statics and dynamics of ATS

effects (e.g., non-termination, exceptions, references), which on the other hand are difficult to properly address in Coq.

The current implementation of ATS [15] is written in Objective Caml, which mainly consists of a type-checker and an interpreter, and a compiler from ATS to C is under active development. The entire implementation (including source code) is made available to the public, and a tutorial is also provided for explaining a variety of language features in ATS. In this paper, we focus on a unique feature in ATS that combines programming with theorem proving, and illustrate by example how this kind of programming paradigm is supported in ATS.

2 Overview of ATS

Some formal syntax of ATS is shown in Figure 1. The language ATS has two components: the static component (statics) which includes types, props and type indices and the dynamic component (dynamics) which includes programs and proof terms. The statics itself is a simply typed language and a type in it is referred to as a *sort*. For instance, we have the following base sorts in ATS: *addr*, *bool*, *int*, *prop*, *type*, *view*, *viewtype*, etc. Static terms L , B , I of sorts *addr*, *bool* and *int* are referred to as static address, boolean and integer terms, respectively. Static terms T of sort *type* are types of program terms, and static terms P of sort *prop*, referred to as props, are types of proof terms. Proof terms exist only to show that their types are inhabited (in order to prove constraints on type indices). Since the type system guarantees that proof functions are total, we may simply erase proof terms after type-checking. We also allow linear proof terms, which are assigned a view V , of sort *view*. Since it is legal to use non-linear proofs as linear ones, we have that *prop* is a subsort of *view* and *type* is a subsort of *viewtype*.

Types, props and views may depend on one or more type indices of static sorts. A special case of such indexed types are singleton types, which are each a type for only one specific value. For instance, $\mathbf{int}(I)$ is a singleton type for the integer equal to I , and $\mathbf{ptr}(L)$ is a singleton type for the pointer that points to the address (or location) L .

We combine proofs with programs using *proving types* of the form $(V \mid T)$ where V and T stand for static terms of sort *view* and *type*, respectively. A proving type formed with a view is assigned the sort *viewtype*; if V can be assigned a prop then we can assign the proving type the sort *type*. We may

	Assigned To	Purity/Linearity
sort	type/prop/view indices	pure
prop	proof terms	pure
type	program terms	effectful
view	linear proof terms	pure, linear
viewtype	program terms with embedded linear proofs	effectful, linear

Fig. 2. Sorts, props, types, views and viewtypes in ATS

think of the proving type $(V \mid T)$ as a refinement of the type T because V often constrains some of the indices appearing in T . For example, the following type:

$$(\mathbf{ADD}(m, n, p) \mid \mathbf{int}(m) * \mathbf{int}(n) * \mathbf{int}(p))$$

is a proving type of sort *type* for a tuple of integers (m, n, p) along with a proof of the prop $\mathbf{ADD}(m, n, p)$ which encodes $m + n = p$ (as is explained later). In the case of props which are linear constraints on integers, or more precisely, constraints on integers that can be transformed into linear integer programming problems, ATS can handle them implicitly, without proofs. Given a linear constraint C and a type T , we have two special forms of types: asserting types of the form $C \wedge T$ and guarded types of the form $C \supset T$. Note that $C \wedge T$ is essentially the proving type $(C \mid T)$, and $C \supset T$ is essentially the type $(C \mid 1) \rightarrow T$, except that assertions and guards are proved and discharged automatically by a built-in decision procedure. Following is an example involving singleton, guarded and asserting types:

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

The meaning of this type should be clear: Each value that can be assigned this type represents a function from nonnegative integers to negative integers.

3 Data-classes in ATS

In ATS it is possible to introduce user defined data-classes which can be sorts, types, props, or views. The intended uses and properties of these classes are described in Figure 2. The syntax for data-class introduction is inspired by that of SML. A new base sort for binary trees of integers can be defined by:

```
datasort itr = leaf | node of (int, itr, itr)
```

Sorts may also be higher order. For example, the sort of higher-order abstract syntax (HOAS) for pure lambda calculus is defined by:

```
datasort tm = ap of (tm, tm) | lm of (tm -> tm)
```

Because there is no recursion in the statics we can allow negative occurrences of the sort being declared without sacrificing strong normalization of static terms.

One may also define datatypes similarly to datatypes in SML. The main difference is that in ATS datatypes may be indexed by static terms and the datatype constructors may universally quantify over static term variables. For example, the following is a datatype for lists which has two indices: one for the type of elements of the list and the other for the length of the list.

```

datatype list (type, int) =
  {a:type} nil (a, 0) | {a:type, n:nat} cons (a, n+1) of (a, list (a, n))

```

Sorts are used as type indices and are often universally quantified over in function and data(prop/type/view/viewtype) definitions. The syntax $\{n:nat\}$ stands for universal quantification of n over the sort nat . Universal quantification can also be guarded by one or more constraints, as in $\{n:int \mid n \geq 0\}$. The sort nat is a subset sort, which is really just a sort with an attached constraint, so $\{n:nat\}$ is equivalent to $\{n:int \mid n \geq 0\}$. Notice that in the definition of the datatype `list`(*type*, *int*), the second index is of sort *int*, yet the constructors can only create terms whose type has a *nat* in that position. The reason for this is that we do not want well-sortedness of statics to depend on solving constraints. So, there is nothing ill-formed about the type `list`(*a*, -1), though it is uninhabited.

Dataprops may be introduced with syntax similar to that for datatypes. For example, the following is a dataprop for proofs that a given integer is in a tree.

```

dataprop ITR(int,itr) = // the first bar (|) is optional
  | {n:int, l:itr, r:itr} ITRbase(n,node(n,l,r))
  | {n:int, n':int, l:itr, r:itr} ITRleft(n,node(n',l,r)) of ITR(n,l)
  | {n:int, n':int, l:itr, r:itr} ITRright(n,node(n',l,r)) of ITR(n,r)

```

This declaration creates three constructors for forming **ITR** proofs:

$$\begin{aligned}
 ITRbase & : \forall n : int. \forall l : itr. \forall r : itr. \mathbf{ITR}(n, node(n, l, r)) \\
 ITRleft & : \forall n : int. \forall n' : int. \forall l : itr. \forall r : itr. \mathbf{ITR}(n, l) \rightarrow \mathbf{ITR}(n, node(n', l, r)) \\
 ITRright & : \forall n : int. \forall n' : int. \forall l : itr. \forall r : itr. \mathbf{ITR}(n, r) \rightarrow \mathbf{ITR}(n, node(n', l, r))
 \end{aligned}$$

In general we construct dataprops in order to prove constraints on the indices of the prop. The proofs are then erased after type-checking, leaving only program terms to be executed. We need to ensure all proof terms are total in order to allow for this erasure semantics. Because of this requirement we restrict the definition of dataprops and dataviews to not have negative occurrences of the prop or view being defined. Without this restriction one could implement a fixed-point operator on props using a dataprop with a negative occurrence. Dataviews are simply linear dataprops and have essentially the same syntax.

4 Programming with Theorem Proving in ATS

In this section, we illustrate by example how the paradigm that combines programming with theorem proving is supported in ATS.

4.1 Programming with Constraints

The design of the concrete syntax of ATS is largely influenced by that of Standard ML (SML) [12]. Previously, we used following concrete syntax to define a datatype constructor `list` for forming a type for a list:

```

datatype list (type, int) =
  {a:type} nil (a, 0) | {a:type, n:nat} cons (a, n+1) of (a, list (a, n))

```

Given two static terms: a type T and an integer I , we can form a datatype $\mathbf{list}(T, I)$ for lists of length I in which each element is of type T . The above concrete syntax indicates that two associated list constructors nil and $cons$ are assigned the following types:

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

The type of $cons$ means that given a value of type T and a list of length I in which each element is of type T , we can construct a list of length $I + 1$ in which each element is of type T . The function that appends two given lists is implemented as follows:

```

fun append {a:type, m:nat, n:nat}
  (xs: list (a, m), ys: list (a, n)): list (a, m+n) =
  case xs of nil () => ys | cons (x, xs') => cons (x, append (xs', ys))

```

where the syntax indicates that $append$ is assigned the following type:

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

That is, $append$ returns a list of length $m + n$ when applied two lists of length m and n , respectively.

When this function is type-checked, two linear constraints on integers are generated. First, when the first given list xs is nil , i.e., the length m is 0, the constraint $n = 0 + n$ is generated in order to assign ys the type $\mathbf{list}(a, m + n)$. Second, in order to assign $cons(x, append(xs', ys))$ the type $\mathbf{list}(a, m + n)$, the constraint $(m - 1) + n + 1 = m + n$ is generated, since xs' is of length $m - 1$, $append(xs', ys)$ is of type $\mathbf{list}(a, (m - 1) + n)$ and $cons(x, append(xs', ys))$ is thus of type $\mathbf{list}(a, (m - 1) + n + 1)$.

In ATS, we also provide a means for the programmer to solve constraints¹ by constructing explicit proofs, supporting a paradigm that combines programming with theorem proving. In Figure 3, we define a datasort $mynat$ with two value constructors: Z for the natural number 0 and S for a successor function on natural numbers. Then given three natural numbers m , n , and p , a dataprop $\mathbf{ADD}(m, n, p)$ represents a proposition $m + n = p$, which is defined inductively. The datatype constructor \mathbf{mylist} now takes a natural number of sort $mynat$ instead of an integer of sort int . The syntax $[p : mynat]$ in the $append$ function, which stands for an existentially quantified static variable, means that there exists a natural number p of sort $mynat$. Now the $append$ function takes two lists of length m and n respectively, and its return type is a proving type of the form $(\mathbf{ADD}(m, n, p) \mid \mathbf{mylist}(a, p))$, meaning that the return value is a list with a

¹ In the current version of ATS, linear constraints on integers can be solved implicitly by a built-in decision procedure based on the approach of Fourier-Motzkin variable elimination [8]. However, the programmer is required to construct explicit proofs to handle nonlinear constraints.

```

datasort mynat = Z | S of mynat

dataprop ADD (mynat, mynat, mynat) =
  | {n:mynat} ADDbas (Z, n, n) // base case
  | {m:mynat, n:mynat, p:mynat} // inductive case
    ADDind (S m, n, S p) of ADD (m, n, p)

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

// '(...) : this syntax is used to form tuples
fun append {a:type, m:mynat, n:mynat}
  (xs: mylist (a, m), ys: mylist (a, n))
  : [p: mynat] '(ADD (m, n, p) | mylist (a, p)) =
  case xs of
  | mynil () => '(ADDbas | ys)
  | mycons (x, xs') => let
    val '(pf | zs) = append (xs', ys)
  in '(ADDind pf | mycons (x, zs)) end

```

Fig. 3. An example of programming with theorem proving

proof that proves the length of the return list is the sum of the lengths of two input lists.

4.2 Programming with Dataprops

In this section we briefly outline a verified call-by-value evaluator for pure λ -calculus. We begin by declaring a static sort to represent λ -terms via higher-order abstract syntax (h.o.a.s.) [13].

```

datasort tm = lm of (tm -> tm) | ap of (tm, tm)

```

This syntax declares a datasort tm with two term constructors: lm which takes as its argument a function of sort $tm \rightarrow tm$, and ap which takes as its arguments two static terms of sort tm . For instance, the λ -term $\lambda x.\lambda y.y(x)$ is represented as $lm (lam x => (lm (lam y => ap (y, x))))$ in the concrete syntax of ATS, where lam is a keyword in ATS for introducing λ -abstraction.

We declare another sort tms to represent environments such that a term of the sort tms consists of a sequence of terms of the sort tm :

```

datasort tms = none | more of (tms, tm)

```

With these two sorts as indices, we can specify the big-step call-by-value evaluation relation as the dataprop **EVAL**:

```

dataprop EVAL(tm, tm, int) =

```

```

| {f:tm -> tm} EVALlam (lm f, lm f, 0) //  $\lambda x.e \rightarrow \lambda x.e$ 
| {t1:tm, t2:tm, f:tm -> tm, v1:tm, v2:tm, n1:nat, n2:nat, n3:nat}
  EVALapp (ap (t1, t2), v2, n1+n2+n3+1) of
    (EVAL (t1, lm f, n1), EVAL (t2, v1, n2), EVAL (f v1, v2, n3))
  // if  $e1 \rightarrow \lambda x.e1'$ ,  $e2 \rightarrow v1$  and  $[v1/x]e1' \rightarrow v2$ , then  $e1(e2) \rightarrow v2$ 

```

```
propdef EVAL0 (t1:tm,t2:tm) = [n:nat] EVAL (t1,t2,n) // prop def.
```

The third index of **EVAL**, an integer, is the length of the reduction sequence and it is needed in forming termination metrics to assure proof totality. Additionally we define **EVAL**₀ to be an **EVAL** of any length. Now we define another prop constructor **ISVAL** expressing that the static term of sort *tm* it is indexed by is a value. Note that λ -abstractions are the only form of values in this simple language.

```
dataprop ISVAL (tm) = {f: tm -> tm} ISVALlam (lm f)
```

Now we can construct two proof functions: *lemma1* which proves that any value evaluates to itself and *lemma2* which proves that any term which is the result of evaluation is a value.

```
prfun lemma1 {t:tm} .< >. (pf :ISVAL (t)): EVAL0 (t, t) =
  case* pf of ISVALlam () => EVALlam
```

```
prfun lemma2 {t1:tm,t2:tm,n:nat} .<n>. (pf: EVAL (t1,t2,n)): ISVAL t2 =
  case* pf of EVALlam () => ISVALlam | EVALapp (_, _, pf') => lemma2 pf'
```

These proof functions make use of two techniques to guarantee totality: the *case** which mandates that the pattern matching following it is exhaustive (an error message is issued otherwise) and the termination metric (written in the form of *<metric>*.) which specifies a well-ordering (based on the lexicographical ordering on tuples of natural numbers) to guarantee termination of the recursion. Now we create datatypes for terms, values and environments:

```
datatype EXP (tms, tm) = ...
```

```
datatype VAL (tm) =
  {ts:tms,f:tm->tm} VALclo(lm f) of (ENV ts, {t:tm} EXP(more(ts,t), f t))
```

```
and ENV (tms) =
  | ENVnil (none)
  | {ts:tms, t:tm} ENVcons (more (ts, t)) of (ISVAL t | ENV ts, VAL t)
```

With these sorts, types and propositions, we can now define an evaluation function whose type proves its correctness:

```
fun eval {ts: tms, t: tm} (env: ENV ts, e: EXP (ts, t))
  : [v:tm] '(EVAL0 (t, v) | VAL v) = ...
```

```
fun evaluate {t: tm} (e: EXP (none, t))
  : [v:tm] '(EVAL0 (t, v) | VAL v) = eval (ENVnil, e)
```

The concrete syntax indicates that the *evaluate* function is assigned the following type:

$$\forall t : tm. \mathbf{EXP}(none, t) \rightarrow \exists v : tm. (\mathbf{EVAL}_0(t, v) \mid \mathbf{VAL}(v))$$

which means that *evaluate* takes a *closed* term t and returns a value v along with a proof that t evaluates to v . Such proofs can be erased before execution leaving a verified interpreter. Please find the omitted code on-line:

<http://www.cs.bu.edu/~hwxi/ATS/EXAMPLE/LF/callByValue.ats>

4.3 Programming with Dataviews

A novel notion of *stateful views* is introduced in ATS to describe memory layouts and reason about memory properties. A stateful view is a form of linear prop, which can also be assigned to proof terms. There is a built-in sort *addr* for static terms L representing memory addresses (or locations). Given a type T and an address L , $T@L$ is a primitive stateful view meaning that a value of the type T is stored at the location L . Given two stateful views V_1 and V_2 , $V_1 \otimes V_2$ is a stateful view that joins V_1 and V_2 together, and $V_1 \multimap V_2$ is a stateful view which yields V_2 when applied to V_1 . We also provide a means to form recursive stateful views through dataviews. For instance, a view for an array is recursively defined below:

```
dataview arrayView (type, int, addr) =
  | {a:type, l:addr} ArrayNone (a, 0, l)
  | {a:type, l:addr, n:nat}
    ArraySome (a, n+1, l) of (a@l, arrayView (a, n, l+1))
```

The stateful view $\mathbf{arrayView}(T, I, L)$ means that there is an array of length I in which all the elements are of type T stored at addresses $L, L+1, \dots, L+(I-1)$. The two associated constructors are of the following props:

$$\begin{aligned} \mathit{ArrayNone} &: \forall a : type. \forall l : addr. \mathbf{arrayView}(a, 0, l) \\ \mathit{ArraySome} &: \forall a : type. \forall l : addr. \forall n : nat. \\ & \quad (a@l \otimes \mathbf{arrayView}(a, n, l+1)) \multimap \mathbf{arrayView}(a, n+1, l) \end{aligned}$$

The types of some built-in functions in ATS can be specified through the use of stateful views. For instance, the types of functions for safe reading from and writing to a pointer are given as follows.

```
dynval getPtr : // read from a pointer
  {a:type, l:addr} (a@l | ptr l) -> (a@l | a)

dynval setPtr : // write to a pointer
  {a1:type, a2:type, l:addr} (a1@l | ptr l, a2) -> (a2@l | unit)
```

According to the type of *getPtr*, the function can be applied a pointer to L only if a proof term of the view $T@L$ is given, which assures that the address L is accessible. In other words, without specifying a proper proof, *getPtr* is not

allowed to be applied. Thus, we can readily prevent dangling pointers from ever being accessed as no proofs for dangling pointers can be provided.

A proof function manipulating views is referred to as a *view change function*. For instance, a view $\mathbf{arrayView}(T, I_0, L)$ can be changed into two views: a view $T@(L + I)$ where $0 \leq I < I_0$ and a functional view $T@(L + I) \multimap \mathbf{arrayView}(T, I_0, L)$ which basically means that given a proof of $T@(L + I)$, it returns a proof of $\mathbf{arrayView}(T, I_0, L)$. This is like taking the I th element out of an array of size I_0 (described by $T@(L + I)$) and leaving the rest (described by $T@(L + I) \multimap \mathbf{arrayView}(T, I_0, L)$). The following view change function *takeOutLemma* implements this idea:

```
prfun takeOutLemma
  {a:type,n:int,i:nat,l:addr | i < n} .<i>. (pf: arrayView (a, n, l))
  : '(a@(l+i), a@(l+i) -o arrayView (a, n, l)) = ...
```

We now introduce an interesting example – an array subscripting function – which makes use of the *takeOutLemma* given above.

```
fun sub {a:type, n:int, i:nat, l:addr | i < n}
  (pf: arrayView (a, n, l) | p: ptr l, i: int i)
  : '(arrayView (a, n, l) | a) = let
    prval '(pf1, pf2) = takeOutLemma {a, n, i, l} (pf)
    val '(pf1 | x) = getPtr (pf1 | p + i)
  in '(pf2 pf1 | x) end
```

In the *sub* function, we use *getPtr* to access the I th element of an array of size I_0 such that $0 \leq I < I_0$ holds. In order to provide a proof of $T@(L + I)$, we apply the view change function *takeOutLemma* to a proof term *pf* of the $\mathbf{arrayView}(T, I_0, L)$. At last it recovers the view for the array using linear function application on two proofs pf_2 and pf_1 , and then returns the value of the type T as we wish. Note that *sub* can be readily compiled into a function with a body of one load instruction after the proofs in *sub* are erased.

4.4 More Examples

Many more interesting and larger examples can be found on-line [15]. In particular, the current library of ATS alone consists of over 20K lines of code written in ATS, where the paradigm of programming with theorem proving is widely employed.

5 Conclusion and Future Work

The language ATS supports a paradigm that combines programming with theorem proving, and we have illustrated by example that how this paradigm is carried out in ATS. In addition to standard non-linear props, a form of linear props, stateful views, is supported in ATS for reasoning about memory. At present, we know no other programming languages that support the paradigm of programming with theorem proving as is described in this paper. In particular,

we emphasize that this paradigm is fundamentally different from the one that extracts programs out of proofs as is advocated in theorem proving systems such as NuPr1 [6] and Coq [2].

We believe that programming with theorem proving is a promising research topic. Developing automated theorem proving (rather than constructing proofs explicitly) and combing it in programming is one of our future works. In order to support more programming paradigms, we plan to combine assembly programming with theorem proving, and employ the idea of view change functions to model the computation of assembly instructions and capture program states (including registers, stacks and heaps) at assembly level. We also plan to reason about other interesting properties by building proper logics on top of ATS. For instance, we would like to develop concurrent programming in ATS, supporting formal reasoning on properties such as deadlocks and race conditions.

References

1. H. P. Barendregt. Lambda Calculi with Types. In S. Abramsky, D. M. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 117–441. Clarendon Press, Oxford, 1992.
2. Y. Bertot and P. 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.
3. C. Chen, R. Shi, and H. Xi. A Typeful Approach to Object-Oriented Programming with Multiple Inheritance. In *Proceedings of the 6th International Symposium on Practical Aspects of Declarative Languages*, Dallas, TX, June 2004. Springer-Verlag LNCS vol. 3057.
4. C. Chen and H. Xi. ATS/LF: a type system for constructing proofs as total functional programs, November 2004. (<http://www.cs.bu.edu/~hwxi/ATS/PAPER/ATSOLF.ps>)
5. C. Chen and H. Xi. Combining Programming with Theorem Proving, November 2004. (<http://www.cs.bu.edu/~hwxi/ATS/PAPER/CPwTP.ps>)
6. R. L. Constable et al. *Implementing Mathematics with the NuPr1 Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
7. R. L. Constable and S. F. Smith. Partial objects in constructive type theory. In *Proceedings of Symposium on Logic in Computer Science*, pages 183–193, Ithaca, New York, June 1987.
8. G. Dantzig and B. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.
9. S. Hayashi and H. Nakano. *PX: A Computational Logic*. The MIT Press, 1988.
10. F. Honsell, I. A. Mason, S. Smith, and C. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, 15 May 1995.
11. N. 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.
12. R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.
13. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23 (7), pages 199–208, Atlanta, Georgia, July 1988. ACM Press.
14. H. Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.
15. H. Xi. Applied Type System, 2005. Available at: <http://www.cs.bu.edu/~hwxi/ATS>.
16. D. Zhu and H. 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.