

# **ATS: a language to make typeful programming real and fun**

Hongwei Xi

Boston University

Work partly funded by NSF grant CCR-0229480

# ATS

ATS is a programming language with a type system rooted in the framework *ATS*. In ATS, a variety of programming paradigms are supported in a typeful manner, including:

- Functional programming (available)
- Imperative programming with pointers (available)
- Object-oriented programming (available)
- Modular programming (available)
- Assembly programming (under development)

Here is the current homepage of ATS:

<http://www.cs.bu.edu/~hwxi/ATS/ATS.html>

# *Applied Type System (ATS)*

- *ATS* is a recently developed framework to facilitate the design and formalization of (advanced) type systems in support of practical programming.
- The name *applied type system* refers to a type system formed in *ATS*, which consists of two components:
  - static component (statics), where types are formed and reasoned about.
  - dynamic component (dynamics), where programs are constructed and evaluated.
- The key salient feature of *ATS*: statics is completely separate from dynamics. In particular, types **cannot** depend on programs.

# *Examples of applied type systems:*

- The simply-typed  $\lambda$ -calculus
- The second-order polymorphic  $\lambda$ -calculus (System  $F$ )
- The higher-order polymorphic  $\lambda$ -calculus (System  $F_\omega$ )
- Dependent ML (DML)
- The second-order polymorphic  $\lambda$ -calculus with guarded recursive types (impredicative formulation)

# *Non-Examples of applied type systems:*

- The dependent  $\lambda$ -calculus ( $\lambda P$ )
- The calculus of constructions ( $\lambda C$ )

# *Syntax for statics*

- The statics is a simply typed language and a type in the statics is referred to as a *sort*. We write  $b$  for a base sort and assume the existence of two special base sorts *type* and *bool*.

sorts  $\sigma ::= b \mid \sigma_1 \rightarrow \sigma_2$

c-sorts  $\sigma_c ::= (\sigma_1, \dots, \sigma_n) \Rightarrow \sigma$

sta. terms  $s ::= a \mid sc(s_1, \dots, s_n) \mid \lambda a : \sigma. s \mid s_1(s_2)$

sta. var. ctx.  $\Sigma ::= \emptyset \mid \Sigma, a : \sigma$

- In practice, we also have base sorts *int* and *addr* for integers and addresses (or locations), respectively. Let us use  $B, I, L$  and  $T$  for static terms of sorts *bool*, *int*, *addr* and *type*, respectively.

# *Some static constants*

<b>1</b>	:	$() \Rightarrow type$
<i>true</i>	:	$() \Rightarrow bool$
<i>false</i>	:	$() \Rightarrow bool$
$\rightarrow$	:	$(type, type) \Rightarrow type$
$\supset$	:	$(bool, type) \Rightarrow type$
$\wedge$	:	$(bool, type) \Rightarrow type$
$\leq$	:	$(type, type) \Rightarrow bool$ (impredicative formulation)

Also, for each sort  $\sigma$ , we assume that the two static constructors  $\forall_\sigma$  and  $\exists_\sigma$  are assigned the sc-sort  $(\sigma \rightarrow type) \Rightarrow type$ .

# *Constraint relation*

A constraint relation is of the following form:

$$\Sigma; \vec{B} \models B$$

where  $\vec{B}$  stands for a sequence of static boolean terms (often referred to as assumptions).

Here is an interesting question:

- Is deduction modulo a special case where  $\vec{B}$  is empty?

# *Some (unfamiliar) forms of types*

- Asserting type:  $B \wedge T$
- Guarded type:  $B \supset T$

Here is an example involving both guarded and asserting types:

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

This type can be assigned to a function from nonnegative integers to negative integers.

# *Syntax for dynamics*

dyn. terms	$d ::= x \mid dc(d_1, \dots, d_n) \mid$ $\mathbf{lam} \ x.d \mid \mathbf{app}(d_1, d_2) \mid$ $\supset^+(v) \mid \supset^-(d) \mid$ $\forall^+(v) \mid \forall^-(d) \mid$ $\wedge(d) \mid \mathbf{let} \ \wedge(x) = d_1 \ \mathbf{in} \ d_2 \mid$ $\exists(d) \mid \mathbf{let} \ \exists(x) = d_1 \ \mathbf{in} \ d_2$
values	$v ::= x \mid dcc(v_1, \dots, v_n) \mid \mathbf{lam} \ x.d \mid$ $\supset^+(v) \mid \forall^+(v) \mid \wedge(v) \mid \exists(v)$
dyn. var. ctx.	$\Delta ::= \emptyset \mid \Delta, x : s$

# *Typing judgment*

A typing judgment is of the following form:

$$\Sigma; \vec{B}; \Delta \vdash d : T$$

# *A datatype declaration in ATS*

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

The concrete syntax means the following:

$$\begin{aligned} \textit{nil} & : \forall a : \textit{type}. \textit{list}(a, 0) \\ \textit{cons} & : \forall a : \textit{type}. \forall n : \textit{int}. \\ & \quad n \geq 0 \supset ((a, \textit{list}(a, n)) \Rightarrow \textit{list}(a, n + 1)) \end{aligned}$$

# *A function declaration in ATS*

```
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))
```

The concrete syntax means that the function *append* is assigned the following type:

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

# *Another function declaration in ATS*

```
fun concat {a:type, m:nat, n:nat}
  (xss: list (list (a, m), n))
  : list (a, m*n) =
case xss of
| nil () => nil
| cons (xs, xss) =>
  append (xs, concat xss)
```

Unfortunately, this code currently **cannot** pass type-checking in ATS because non-linear constraints on integers are involved.

# *Programming with theorem proving*

- We introduce a new sort *prop* into the statics and use  $P$  for static terms of sort *prop*, which are often referred to as props.
- A prop is like a type, which is intended to be assigned to special dynamic terms that we refer to as proof terms.
- A proof term is required to be pure and total, and it is to be erased before program execution. In particular, we do not extract programs out of proofs. Consequently, we can and do construct classical proofs.

# *A dataprop declaration in ATS*

```
dataprop MUL (int, int, int) =  
  | {n:int} MULbas (0, n, 0)  
  | {m:nat, n:int, p:int}  
    MULind (m+1, n, p+n) of MUL (m, n, p)  
  | {m:pos, n:int, p:int}  
    MULneg (~m, n, ~p) of MUL (m, n, p)
```

The concrete syntax means the following:

$$\begin{aligned}0 * n &= 0 \\(m + 1) * n &= m * n + n \\(-m) * n &= -(m * n)\end{aligned}$$

# A proof function declaration in ATS

```
prfun lemma {m:nat, n:nat, p:int} .<m>.
  (pf: MUL (m, n, p)): [p >= 0] prunit =
  case* pf of
  | MULbas () => '()
  | MULind pf' =>
    let prval _ = lemma pf' in '() end
```

The proof function proves:

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

We need to verify that *lemma* is a total function:

- $\langle m \rangle$  is a termination metric.
- *case\** requires pattern matching to be exhaustive.

# *An example of programming with theorem proving*

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

**Remark** Proofs are completely erased before program execution. In other words, there is no proof construction at run-time.

# *Stateful views*

A stateful view is a linear prop (not a linear type).

- 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  $V_2$ , we use  $V_1 \otimes V_2$  for a stateful view that joins  $V_1$  and  $V_2$  together.
- We also provide a means for forming recursive stateful views.

# *A dataview declaration in ATS*

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

The concrete syntax means the following:

*ArrayNone* :  $\forall a : \text{type} . \forall l : \text{addr} . \text{arrayView}(a, 0, l)$

*ArraySome* :  $\forall a : \text{type} . \forall n : \text{nat} . \forall l : \text{addr} .$   
 $(a @ l, \text{arrayView}(a, n, l + 1)) \rightarrow$   
 $\text{arrayView}(a, n + 1, l)$

# *Some built-in functions*

```
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)
```

*getPtr* :  $\forall a : \text{type} . \forall l : \text{addr} . (a@l, \mathbf{ptr}(l)) \rightarrow (a@l, a)$

*setPtr* :  $\forall a_1 : \text{type} . \forall a_2 : \text{type} . \forall l : \text{addr} .$   
 $(a_1@l, \mathbf{ptr}(l), a_2) \rightarrow (a_2@l, \mathbf{1})$

# *Viewtypes*

A viewtype is a linear type.

viewtypes  $VT ::= T \mid V * VT$

The intuition is that the construction of values of viewtypes may consume resources.

# Accessing the first element of an array

```
fun getFirst {a:type, n:pos, l:addr}
  (pf: arrayView (a, n, l) | p: ptr l)
: '(arrayView (a, n, l) | a) =
let
  prval ArraySome (pf1, pf2) = pf
  val '(pf1 | x) = getPtr (pf1 | p)
in
  '(ArraySome (pf1, pf2) | x)
end
```

*getFirst* :  $\forall a : \text{type} . \forall n : \text{int} . \forall l : \text{addr} . n > 0 \supset$   
 $(\text{arrayView}(a, n, l), \mathbf{ptr}(l)) \rightarrow \text{arrayView}(a, n, l) * a$

# *A proof function for view change*

```
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)) =
let
  prval ArraySome (pf1, pf2) = pf
in
  sif i > 0 then
    let
      prval '(pf21, pf22) =
        takeOutLemma {a, n-1, i-1, l+1} (pf2)
      in
        '(pf21, llam pf21 => ArraySome(pf1, pf22 pf21))
    end
  else '(pf1, llam pf1 => ArraySome (pf1, pf2))
end
```

# *Subscripting an array*

```
// introducing a type definition
typedef natLt (n: int) = [a: nat | a < n] int n

// implementing array subscription
fun get {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 padd i)
in
  '(pf2 pf1 | x)
end
```

# *Persistent stateful views*

Given an ephemeral stateful view  $V$ , we can form a boxed persistent view  $\Box V$ . In concrete syntax, we use  $!$  for  $\Box$ .

- First and foremost, there is no relation between  $\Box$  and the modal operator  $!$  in linear logic.
- A function is allowed to make use of a boxed view  $\Box V$  only if it treats  $V$  as an invariant, that is, the type of the function is of the following form:

$$V * VT \rightarrow V * VT'$$

- For instance, *getPtr* is assigned the following type

$$\forall a : type. \forall l : addr. (a@l, \mathbf{ptr}(l)) \rightarrow (a@l, a)$$

which indicates that *getPtr* treats  $a@l$  as an invariant.

# *Implementing references (1)*

```
typedef ref (a: type) =  
  [l:addr] '(! (a@l) | ptr l)
```

```
dynval getRef : {a:type} ref a -> a
```

```
dynval setRef : {a:type} (a, ref a) -> unit
```

# *Implementing references (2)*

```
fun getPtr0 {a:type, l:addr}  
  (pf: a @ l | (*none*) | p: ptr l)  
  : a = getPtr (pf | p)
```

```
fun getRef {a:type} (r: ref a): a =  
  let  
    val '(pf | p) = r  
  in  
    getPtr0 (pf | (*none*) | p)  
end
```

# *Implementing references (3)*

```
fun setPtr0 {a:type, l:addr}
  (pf: a @ l | (*none*) | p: ptr l, x: a)
  : unit = setPtr (pf | p, x)

fun setRef {a:type} (r: ref a, x: a): unit =
  let
    val '(pf | p) = r
  in
    setPtr0 (pf | (*none*) | p, x)
  end
```

# Time for a demo

- The demo is about two implementations of functional lists ...

# *Related work*

Here is only a fraction:

- Theorem proving systems: NuPrl, Coq, ...
- (Meta) Logical Frameworks: Twelf, ...
- Functional Languages: Delphin, Omega, Vera, ...
- Dependently Typed Functional Languages: Cayenne, Dependent ML, Epigram, ...
- Separation logic, ...
- Vault, Effective theory of refinements,  $L^3$ , ...

# *Conclusion and future directions*

- We have outlined a design to support programming with theorem proving.
- In addition, we have carried out this design in the programming language ATS.
- We are currently also keen to formally support reasoning on properties such as deadlocks and race conditions. After all, multi-threaded programming is simply indispensable in general software practice.