A Brief Introduction to ATS

Hongwei Xi

Boston University

Work partly funded by NSF

ATS(1)

- ATS is a statically typed programming language designed to unify implementation with formal specification.
- The type system of ATS is rooted in the framework *Applied Type System*, which gives the language its name.
- Both dependent types (DML-style) and linear types are available in ATS.
- The current compiler of ATS (ATS/Anairiats) is written in ATS itself, consisting of approximately 100K lines of code.

ATS(2)

- The core of ATS is a call-by-value functional programming language.
- ATS supports a programming paradigm referred to as *programming with theorem-proving (PwTP)*, allowing code for run-time computation and code for proof construction to be written in a syntactically intertwined manner.
- Imperative programming in ATS makes essential use of PwTP.
- ATS and C share the same native data representation, and programs in ATS are compiled directly into C code. To some extent, ATS can be thought of as a front end for C.
- For more details, please visit:

`http://www.ats-lang.org`

Applied Type System (ATS)

- *ATS* is a framework developed to facilitate designing and formalizing (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* is that statics is completely separate from dynamics. In particular, types cannot depend on programs.

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	σ	$::=$	$b \mid \sigma_1 \rightarrow \sigma_2$
c-sorts	σ_c	$::=$	$(\sigma_1, \ldots . \sigma_n) \Rightarrow \sigma$
sta.cd terms	s	$::=$	$a \mid sc(s_1, \ldots, s_n) \mid \lambda a : \sigma . s \mid s_1(s_2)$
sta. var. ctx.	Σ	$::=$	$\emptyset \mid \Sigma, a : \sigma$

- 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 the sorts *bool*, *int*, *addr* and *type*, respectively.

Some Static Constants

$$
\begin{array}{rcll}
\mathbf{1} & : & () \Rightarrow type & \\
true & : & () \Rightarrow bool & \\
false & : & () \Rightarrow bool & \\
\rightarrow & : & (type, type) \Rightarrow type & \\
\supset & : & (bool, type) \Rightarrow type & \\
\wedge & : & (bool, type) \Rightarrow type & \\
\leq & : & (type, type) \Rightarrow bool & \text{(impredicative formulation)}
\end{array}
$$

Also, for each sort σ, 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). This relation holds if one of the assumptions in $\vec{B}$ is false or the conclusion B is true.

A Sample Constraint

The following constraint is generated when an implementation of binary search on arrays is type-checked:

$$\Sigma; \vec{B} \models l + (h - l)/2 + 1 \leq sz$$

where

$$\begin{aligned} \Sigma &= h : int, l : int, sz : int \\ \vec{B} &= l \geq 0, sz \geq 0, 0 \leq h + 1, h + 1 \leq sz, 0 \leq l, l \leq sz, h \geq l \end{aligned}$$

We may employ linear integer programming to solve such a constraint.

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:

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

This type can be assigned to a function from nonnegative integers to negative integers. As a probably more interesting example, the usual run-time assertion function can be given the following type:

$$\forall a : bool.\ \mathbf{bool}(a) \rightarrow (a = true) \wedge \mathbf{1}$$

Syntax for Dynamics

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

Typing Judgment

A typing judgment is of the following form:

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

where

Σ	:	static variable context
$\vec{B}$	:	assumption set
Δ	:	dynamic variable context
d	:	dynamic term
T	:	type

Two Function Definitions in ATS

```
fun fact (x: int): int =
  if x = 0 then 1 else x * fact (x-1)
// end of [fact]

val fact10 = fact (10) // this call can typecheck
val fact_1 = fact (~1) // this call does typecheck

fun fact2 {n:nat}
  (x: int n): int =
  if x = 0 then 1 else x * fact2 (x-1)
// end of [fact2]

val fact10 = fact2 (10) // this call can typecheck
val fact_1 = fact2 (~1) // this call does not typecheck
```

A Datatype Declaration in ATS

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

The concrete syntax means the following:

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

Programming with Theorem-Proving

- We introduce a new sort *prop* into the statics and use P for static terms of the 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

```
datatype FIB (int, int) =
  | FIB0 (0, 0) of () // fib(0) = 0
  | FIB1 (1, 1) of () // fib(1) = 1
  | {n:nat}{r0,r1:nat} // fib(n+2) = fib(n)+fib(n+1)
    FIB2 (n+2, r0+r1) of (FIB (n, r0), FIB (n+1, r1))
```

The following function *fib* computes Fibonacci numbers:

```
fun fib {n:nat}
  (n: int (n)): [r:nat] (FIB (n, r) | int (r))
// end of [fib]
```

A Direct Implementaation of fib

```
implement fib (n) =
  case+ n of
  | 0 => (FIB0 () |  0)
  | 1 => (FIB1 () |  1)
  | _ =>> let
      val (pf0 | r0) = fib (n-2)
      val (pf1 | r1) = fib (n-1)
    in
      (FIB2 (pf0, pf1) | r0 + r1)
    end
// end of [fib]
```

Views

A view is a linear prop.

- Given a type T and an address L, $T@L$ is a primitive view meaning that a value of the type T is stored at the location L.
- Given two views V_1 and V_2, we use $V_1 \otimes V_2$ for a view that joins V_1 and V_2 together.
- We also provide a means for forming recursive views.

Some Built-in Functions

```
fun{a:type}
ptrget {l:addr}
  (pf: a@l | p: ptr l): (a@l | a)
// end of [ptrget]

fun{a:type}
ptrset {l:addr}
  (pf: (a?)@l | p: ptr l, x: a): (a@l | void)
// end of [ptrset]
```

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

$$ptrset \quad : \quad \forall a : type. \forall l : addr.\ (a?@l \mid \mathbf{ptr}(l), a) \rightarrow (a@l \mid \mathbf{1})$$

Example: Swap

```
fun{a:type}
swap {l1,l2:addr} (
  pf1: a@l1, pf2: a@l2 | p1: ptr l1, p2: ptr l2
) : (a@l1, a@l2 | void) = let
  val (pf1 | x1) = ptrget<a> (pf1 | p1)
  val (pf2 | x2) = ptrget<a> (pf2 | p2)
  val (pf1 | ()) = ptrset<a> (pf1 | p1, x2)
  val (pf2 | ()) = ptrset<a> (pf2 | p2, x1)
in
  (pf1, pf2 | ())
end // end of [swap]
```

Syntactical Convenience

```
fun{a:type}
ptrget {l:addr} (pf: !a@l | p: ptr l): a

fun{a:type}
ptrset {l:addr}
  (pf: !(a?)@l >> a@l | p: ptr l, x: a): void
// end of [ptrset]
```

Another Implementation of Swap

```
fun{a:type}
swap {l1,l2:addr} (
  pf1: !a@l1, pf2: !a@l2
| p1: ptr l1, p2: ptr l2
) : void = let
  val tmp = !p1 in !p1 := !p2; !p2 := tmp
end // end of [swap]
```

A Dataview Declaration in ATS

```
dataview array_v (a:type, int, addr) =
  | {l:addr}
    array_v_nil (a,0,l)
  | {n:nat}{l:addr}
    array_v_cons (a,n+1,l) of (a@l, array_v (a,n,l+1))
```

The concrete syntax means the following:

$$
\begin{aligned}
\mathit{array\_v\_nil} &: \forall a : \mathit{type}.\forall l : \mathit{addr}.\ \ \mathit{array\_v}(a,0,l) \\
\mathit{array\_v\_cons} &: \forall a : \mathit{type}.\forall n : \mathit{nat}.\forall l : \mathit{addr}. \\
&: \quad (a@l, \mathit{array\_v}(a,n,l+1)) \rightarrow \mathit{array\_v}(a,n+1,l)
\end{aligned}
$$

Proof Functions for View Change

```
prfun split
  {a:type}
  {n:int}{i:nat | i <= n}
  {l:addr} (
  pf: array_v (a, n, l)
) : (array_v (a, i, l), array_v (a, n-i, l+i))

prfun unsplit
  {a:type}
  {n1,n2:nat}{l:addr} (
  pf1: array_v (a, n1, l), pf2: array_v (a, n2, l+n1)
) : array_v (a, n1+n2, l)
```

Another Proof Function for View Change

```
prfun takeout
  {a:type}
  {n:int}{i:nat | i < n}
  {l:addr} (
  pf: array_v (a, n, l)
) : (a @ l+i, a @ l+i -<lin> array_v (a, n, l))
```

Array Subscripting

```
fun{a:type}
arrsub
  {n:int}{i:nat | i < n}
  {l:addr} (
  pf: !array_v (a, n, l) | p: ptr l, i: int i
) : a = x where {
  prval (pfat, fpf) = takeout {a}{n}{i} (pf)
  val x = ptrget<a> (pfat | p+i)
  prval () = pf := fpf (pfat)
} // end of [arrsub]
```

Viewtypes

- A linear type in ATS is given the name viewtype, which is chosen to indicate that a linear type consists of two parts: one part for views and the other for types. We use *viewtype* as the sort for viewtypes.
- Given a view V and a type T, the tuple (V | T) is a viewtype, where the bar symbol (|) is a separator (just like a comma) to separate views from types.
- What seems a bit surprising is the opposite: For each viewtype VT, we may assume the existence of a view V and a type T such that VT is equivalent to (V | T). Formally, this T can be referred as VT?! in ATS. This somewhat unexpected interpretation of linear types stresses that the linearity of a viewtype comes entirely from the view part residing within it.

Generic Types for ptrget and ptrset

```
fun{a:viewtype}
ptrget {l:addr}
  (pf: !a@l>> (a?!)@l | p: ptr l): a
// end of [ptrget]

fun{a:viewtype}
ptrset {l:addr}
  (pf: !(a?)@l >> a@l | p: ptr l, x: a): void
// end of [ptrset]
```

Example: Viewtype for Linear Closures

Given two types A and B, a pointer to some address L where a closure function is stored that takes a value of the type A to return a value of the type B can be given the viewtype cloptr(A, B, L):

```
viewtypedef cloptr
  (a:t@ype, b:t@ype, l:addr) =
  [env:t@ype] (((&env, a) -> b, env) @ l | ptr l)
// end of [cloptr]
```

In the function type `(&env, a) -> b`, the symbol & indicates that the corresponding function argument is passed by reference, that is, the argument is required to be a left-value and what is actually passed is the address of the left-value.

A Dataviewtype Declaration in ATS

```
dataviewtype
list_vt (a:type, int) =
  | {n:nat}
    list_vt_cons (a,n+1) of (a, list_vt (a,n))
  | list_vt_nil (a, 0)
```

The concrete syntax means the following:

$$
\begin{aligned}
\mathit{list\_vt\_nil} &: \forall a : \mathit{type}.\ \mathit{list\_vt}(a, 0) \\
\mathit{list\_vt\_cons} &: \forall a : \mathit{type}. \forall n : \mathit{nat}. \\
&\qquad (a, \mathit{list\_vt}(a, n)) \rightarrow \mathit{list\_vt}(a, n+1)
\end{aligned}
$$

Example: Reverse

```
fun{a:type}
reverse {n:nat} (
  xs: list_vt (a, n)
) : list_vt (a, n) = revapp (xs, list_vt_nil)
```

Example: Reverse-Append

```
fun{a:type}
revapp {m,n:nat} (
  xs: list_vt (a, m), ys: list_vt (a, n)
) : list_vt (a, m+n) =
  case+ xs of
  | list_vt_cons
      (_, !ptl) => let
      val tl = !ptl; val () = !ptl := ys
      prval () = fold@ (xs)
    in
      revapp (tl, xs)
    end
  | ~list_vt_nil () => ys
// end of [revapp]
```

Abstract Views

```
absview free_v (n:int, l:addr)

fun free {n:nat}{l:addr} (
  pfgc: free_v (n, l), pfat: bytes(n) @ l | p: ptr l
) : void // end of [free]

dataview malloc_v (n:int, addr) =
  | {l:agz} malloc_v_succ (n, l) of
      (free_v (n, l), bytes(n) @ l | ptr l)
  | malloc_v_fail (n, null) of ()

fun malloc {n:nat}
  (n: size_t (n)): [l:addr] (malloc_v (n, l) | ptr l)
// end of [malloc]
```

Abstract Viewtypes (1)

```
absviewtype queue (a:viewtype, n:int)

fun{a:viewtype} queue_new (): queue (a, 0)
fun{a:viewtype} queue_free (obj: queue (a, 0)): void

fun{a:viewtype}
queue_enque {n:nat} (
  obj: !queue (a, n) >> queue (a, n+1), x: a
) : void // end of [queue_enque]

fun{a:viewtype}
queue_deque {n:pos}
  (obj: !queue (a, n) >> queue (a, n-1)): a
```

Abstract Viewtypes (2)

```
absviewtype mylock (v:view)

fun mylock_create
  {v:view} (pf: v | (*none*)): mylock(v)

fun mylock_destroy
  {v:view} (lock: mylock v): (v | void)

fun mylock_acquire
  {v:view} (lock: !mylock v): (v | void)

fun mylock_release
  {v:view} (pf: v | lock: !mylock v): void
```

Conclusion

- ATS is a statically type programming language that unifies implementation with (a form of) formal specification.
- The signatory feature of ATS is a programming paradigm referred to as programming-with-theorem-proving (PwTP) in which code for (run-time) computation and code for proof construction can be written in a syntactically intertwined manner.
- Views are linear props and viewtypes combine views with types. In particular, the linearity of a viewtype comes entirely from its view part. Imperative programming in ATS is built on top of views and viewtypes (together with the support of PwTP).

Future Direction

- Applying ATS to systems programming. We need more case studies.
- Reducing the amount of “administrative code” needed for proof manipulation. We want to investigate more effective approaches to proof management in order to better support PwTP.
- Identifying the potential of PwTP in areas other than systems programming (e.g., security).

The End

Thank you!