To memory safety through proofs and beyond

Hongwei Xi

Boston University

Work partly funded by NSF grant CCR-0229480

Overview of the talk

- Introduction to ATS
- Combining Programming with Theorem Proving
- Introduction to Stateful Views
- Ascribing Types to library functions in C
- Conclusion

ATS

ATS is a programming language with a type system rooted in the framework Applied Type System ($\mathcal{ATS}$). 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 ($\mathcal{ATS}$)

- $\mathcal{ATS}$ is a framework developed 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 $\mathcal{ATS}$, 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 $\mathcal{ATS}$: statics is completely separate from dynamics. In particular, types cannot depend on programs.

Examples of applied type systems:

- The simply-typed λ-calculus
- The second-order polymorphic λ-calculus (System F)
- The higher-order polymorphic λ-calculus (System F_ω)
- Dependent ML (DML)
- The second-order polymorphic λ-calculus with guarded recursive types (impredicative formulation)

Non-examples of applied type systems:

- The dependent λ-calculus (λP)
- The calculus of constructions (λC)

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. 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 sorts $bool$, int, $addr$ and $type$, respectively.

Some static constants

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

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

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).

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

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

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:

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

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:

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

Syntax for dynamics

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

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

The subtyping rule

We have the following subtyping rule:

$$\frac{\Sigma; \vec{B}; \Delta \vdash d : T \quad \Sigma; \vec{B} \models T \leq T'}{\Sigma; \vec{B}; \Delta \vdash d : T'}$$

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:

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

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:

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

Overview of the rest of the talk

- Combining Programming with Theorem Proving
- Introduction to Stateful Views
- Ascribing Types to library functions in C
- Conclusion

Another function declaration in ATS

```
fun concat {a:type, m:nat, n:nat}
   (xss: list (list (a, m), n))
  : list (a, m*n) = // m*n is non-linear
  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 captures the following definition:

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

A proof function declaration in ATS

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

The proof function establishes:

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

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

- $\langle m \rangle$ is a termination metric.
- *case*$\star$ 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.

Overview of the rest of the talk

- Introduction to Stateful Views
- Ascribing Types to library functions in C
- Conclusion

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

The concrete syntax means the following:

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

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)
```

$$\begin{array}{lcl} \mathit{getPtr} & : & \forall a : \mathit{type}.\forall l : \mathit{addr}.\ \ (a@l \mid \mathbf{ptr}(l)) \rightarrow (a@l \mid a) \\ \mathit{setPtr} & : & \forall a_1 : \mathit{type}.\forall a_2 : \mathit{type}.\forall l : \mathit{addr}. \\ & & \qquad (a_1@l \mid \mathbf{ptr}(l), a_2) \rightarrow (a_2@l \mid \mathbf{1}) \end{array}$$

Viewtypes

A viewtype is a linear type.

$$\text{viewtypes} \quad VT \quad := \quad T \mid (V \mid VT)$$

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:int, l:addr | n > 0}
   (pf: array_v (a, n, l) | p: ptr l)
  : ’(array_v (a, n, l) | a) =
  let
     prval ArraySome (pf1, pf2) = pf
     val ’(pf1 | x) = getPtr (pf1 | p)
  in
     ’(ArraySome (pf1, pf2) | x)
  end
```

$$\begin{aligned} getFirst \quad : \quad & \forall a : type.\forall n : int.\forall l : addr.n > 0 \supset \\ & (array\_v(a, n, l) \mid \mathbf{ptr}(l)) \rightarrow (array\_v(a, n, l) \mid a) \end{aligned}$$

A proof function for view change (1)

Lemma (Takeout) *Assume $array\_v(T, I_0, L)$ for some T, I_0, L. If I_1 is an integer satisfying $0 \leq I_1 < I_0$, then we have $T@L + I_1$ and $T@L + I_1 \multimap array\_v(T, I, L)$.*

Proof: We proceed by induction on I_1.

- $I_1 = 0$. $\ldots$
- $I_1 > 0$. Clearly, $I_0 > 0$ holds. So we have $T@L$ and $array\_v(T, I_0 - 1, L + 1)$. By induction hypothesis, we have $T@L + 1 + (I_1 - 1)$, which is $T@L + I_1$, and $T@L + I_1 \multimap array\_v(T, I_0 - 1, L + 1)$, which yields $T@L + I_1 \multimap array\_v(T, I_0, L + 1)$ when combined with $T@L$.

A proof function for view change (2)

```
prfun takeOutLemma
     {a:type, n:int, i:nat, l:addr | i < n} .<i>.
     (pf: array_v (a, n, l))
    : ’(a @ l+i, a @ l+i -o array_v (a, n, l)) =
  let
     prval ArraySome (pf1, pf2) = pf
  in
     sif i > 0 then
       let
          prval ’(pf21, pf22) = // induction hypothesis
            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

```
// implementing array subscription
fun get {a:type, n:int, i:nat, l:addr | i < n}
   (pf: array_v (a, n, l) | p: ptr l, i: int i)
  : ’(array_v (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
```

Overview of the talk

- Ascribing ATS Types to library functions in C
- Conclusion

Ascribing types to C library functions

By ascribing types in ATS to C library functions, we expect to facilitate safer and securer programming with these functions in ATS.

Ascribing types to malloc and free (1)

The view $byte\_arr\_v(I, L)$ means that there are I consecutive bytes of memory available that starts at the address L.

```
free :
  {n:nat,l:addr}
    (byte_arr_v (n, l) | ptr l) -> unit

malloc :
  {n:nat} int n ->
    [l:addr] (byte_arr_v (n, l) | ptr l)
```

Ascribing types to malloc and free (2)

The view $\mathit{free}\_v(I, L)$ is abstract. Intutitively, it means that the I consecutive bytes of memory starting at address L can be freed *if* they are available.

```
free :
  {n:nat,l:addr}
    (free_v(n,l), byte_arr_v (n,l) |
     ptr l) -> unit

malloc :
  {n:nat} int n ->
    [l:addr]
      ’(free_v(n,l), byte_arr_v(n,l) | ptr l)
```

Ascribing types to malloc and free (3)

```
dataview malloc_v (int, addr) =
  | {n:nat} malloc_v_fail (n, null)
  | {n:nat, l:addr | l <> null}
       malloc_v_succ (n, l) of
         (free_v (n, l), byte_arr_v (n, l))
```

Given an integer I and an address L, a proof of the view $malloc\_v(I, L)$ can be turned into a proof of the empty view if L is the null pointer, or it can be turned two proofs of the views $free\_v(I, L)$ and $byte\_arr\_v(I, L)$, respectively, if L is not the null pointer.

```
malloc : {n:nat} int n ->
  [l:addr] ’(malloc_v (n, l) | ptr l)
```

Ascribing types to malloc and free (4)

```
fun malloc_exn {n:nat} (n: int n)
  : [l:addr]
      ’(free_v (n,l), byte_arr_v (n,l) |
        ptr l) =
  let val ’(pf | p) = malloc (n) in
     if p <> null then let
        prval malloc_v_succ (pf1, pf2) = pf
     in
       ’(pf1, pf2 | p)
     end else let
        prval malloc_v_fail () = pf
     in
        raise MemoryAllocException ()
     end
  end
```

Ascribing types to fopen and fclose (1)

Here are the types of $fopen$ and $fclose$ in C:

```
FILE *fopen(char *path, char *mode);

int fclose( FILE *stream);
```

Ascribing types to fopen and fclose (2)

```
absview FILE_v (addr)
typedef FILE = [l:addr] ’(FILE_v l | ptr l)

dataview fopen_v (addr) =
  | fopen_v_fail (null)
  | {l:addr | l <> null}
      fopen_v_succ (l) of FILE_v l

fopen : (String, String) ->
  [l:addr] ’(fopen_v l | ptr l)

fclose : {l:addr} (FILE_v l | ptr l) -> Int
```

Can we also handle $fcloseall$?

Yes, we can, but it is a long story ...

Related work

Here is only a fraction:

- Theorem proving systems: NuPrl, Coq, Isabelle/HOL, PVS, ...
- (Meta) Logical Frameworks: Twelf, ...
- Dependently Typed Functional Langauges: Cayenne, Dependent ML, Delphin, Epigram, Omega, Vera, ...
- Alias types, Separation logic, ...
- Vault, Effective theory of refinements, L^3, ...

Conclusion and future directions

- A design is outlined to support programming with theorem proving. In particular, we have shown how this design can be used to guarantee memory safety in the presence of pointers and pointer arithmetic.
- This design is carried out in the programming language ATS, which is freely available to the public.
- It is clearly desirable to formally support reasoning on properties such as deadlocks and race conditions. After all, multi-threaded programming is simply indispensable in general software practice.