To Memory Safety through Proofs

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 Applied Type System (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)
Current Status of ATS

- The current implementation of ATS is done in O’Caml, including a type-checker, an interpreter and a compiler from ATS to C.

- The run-time system of ATS supports untagged native data representation (in addition to tagged data representation) and a conservative GC.

- The library of ATS is done in ATS itself, consisting of over 25k lines of code.

For more information, the current homepage of ATS is available at:

http://www.cs.bu.edu/~hwxi/ATS
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.
The types of read and write

\[ \text{get\_ptr} : \quad \text{// read from a pointer} \]
\[ \{a:\text{type}, \ l:\text{addr}\} \ (a@l \mid \text{ptr}\ l) \rightarrow (a@l \mid a) \]

\[ \text{set\_ptr} : \quad \text{// write to a pointer} \]
\[ \{a_1:\text{type}, \ a_2:\text{type}, \ l:\text{addr}\} \]
\[ (a_1@l \mid \text{ptr}\ l, \ a_2) \rightarrow (a_2@l \mid \text{unit}) \]

\[ \text{get\_ptr} : \quad \forall a : \text{type} . \forall l : \text{addr} . \ (a@l \mid \text{ptr}(l)) \rightarrow (a@l \mid a) \]
\[ \text{set\_ptr} : \quad \forall a_1 : \text{type} . \forall a_2 : \text{type} . \forall l : \text{addr} . \]
\[ (a_1@l \mid \text{ptr}(l), \ a_2) \rightarrow (a_2@l \mid \text{unit}) \]
fun swap 
  {a1:type, a2:type, l1:addr, l2:addr} 
  (pf1: a1 @ l1, pf2: a2 @ l2 | 
   p1: ptr l1, p2: ptr l2) 
  : '(a2 @ l1, a1 @ l2 | unit) 
let 
  val '(pf3 | tmp1) = get_ptr (pf1 | p1) 
  val '(pf4 | tmp2) = get_ptr (pf2 | p2) 
  val '(pf5 | _) = set_ptr (pf3 | p1, tmp2) 
  val '(pf6 | _) = set_ptr (pf4 | p2, tmp1) 
in 
  '(pf5, pf6 | '()) 
end
A dataview declaration in ATS

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

The concrete syntax means the following:

array_v_none : \forall a : type. \forall l : addr. array_v(a, 0, l)
array_v_some : \forall a : type. \forall n : nat. \forall l : addr.
              (a@l, array_v(a, n, l + 1)) \rightarrow
              array_v(a, n + 1, l)
A viewtype is a linear type.

\[
\text{viewtypes} \quad VT \quad ::= \quad T \mid V \ast VT
\]

The intuition is that the construction of values of viewtypes may consume resources.
Accessing the first element of an array

fun get_first {a:type, n:pos, l:addr}
  (pf: array_v (a, n, l) | p: ptr l)
  : '(array_v (a, n, l) | a) =
  let
      prval array_v_some (pf1, pf2) = pf
      val '(pf1' | x) = get_ptr (pf1 | p)
    in
      '(array_v_some (pf1', pf2) | x)
  end

get_first : ∀a : type.∀n : int.∀l : addr.n > 0 ⊃
          (array_v(a, n, l) | ptr(l)) → (array_v(a, n, l) | a)
A proof function for view change

prfun take_out_lemma
  \{a: type, n: int, i: nat, l: addr \mid i < n\} .<i>.
  (pf: array_v (a, n, l))
  : '(a @ l+i, a @ l+i -o array_v (a, n, l)) =
let prval array_v_some (pf1, pf2) = pf in
sif i > 0 then
  let
    prval '(pf21, pf22) =
      take_out_lemma \{a, n-1, i-1, l+1\} (pf2)
  in
    '(pf21, llam pf21 =>
      array_v_some(pf1, pf22 pf21))
  end
else '(pf1, llam pf1 => array_v_some (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: array_v (a, n, l) | p: ptr l, i: int i)
  : '(array_v (a, n, l) | a) =
  let
    prval '(pf1, pf2) = take_out_lemma {a,n,i,l} (pf)
    val '(pf1 | x) = get_ptr (pf1 | p padd i)
  in
    '(pf2 pf1 | x)
  end
Ascribing types to C library functions

By ascribing types in ATS to C library functions, we expect to facilitate safe programming with these functions in ATS.
Ascribing types to \texttt{malloc} and \texttt{free} (1)

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

\begin{align*}
\text{free} : \\
\{n : \text{nat}, l : \text{addr}\} \\
(\text{byte\_arr\_v}(n, l) \mid \text{ptr } l) \rightarrow \text{unit}
\end{align*}

\begin{align*}
\text{malloc} : \\
\{n : \text{nat}\} \text{ int } n \rightarrow \\
[l : \text{addr}] (\text{byte\_arr\_v}(n, l) \mid \text{ptr } l)
\end{align*}
Ascribing types to \texttt{malloc} and \texttt{free} (2)

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

\begin{align*}
\text{free} : & \quad \{n : \text{nat}, l : \text{addr}\} \\
& \quad (\text{free}_v(n, l), \text{byte}_\text{arr}_v(n, l) \mid \text{ptr } l) \rightarrow \text{unit} \\
\text{malloc} : & \quad \{n : \text{nat}\} \text{ int } n \rightarrow \\
& \quad [l : \text{addr}] \\
& \quad \left(\text{free}_v(n, l), \text{byte}_\text{arr}_v(n, l) \mid \text{ptr } l\right)
\end{align*}
Ascribing types to `malloc` and `free` (3)

\[
dataview \text{malloc\_v} \ (\text{int}, \ \text{addr}) =
\begin{align*}
| \{n: \text{nat}\} & \text{malloc\_v\_fail} \ (n, \ \text{null}) \\
| \{n: \text{nat}, \ l: \text{addr} \ | \ l \nless null\} & \text{malloc\_v\_succ} \ (n, \ l) \text{ of} \\
(\text{free\_v} \ (n, \ l), \ \text{byte\_arr\_v} \ (n, \ l))
\end{align*}
\]

Given an integer \(I\) and an address \(L\), a proof of the view \(\text{malloc\_v}(I, L)\) 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 \(\text{free\_v}(I, L)\) and \(\text{byte\_arr\_v}(I, L)\), respectively, if \(L\) is not the null pointer.

\[
\text{malloc} : \ \{n: \text{nat}\} \ \text{int} \ n \rightarrow \\
[\text{l: addr} \ ' \ (\text{malloc\_v} \ (n, \ l) \ \mid \ \text{ptr} \ l)
\]
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
Ascribing types to `fopen` and `fclose` (1)

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

```c
FILE *fopen(char *path, char *mode);
int fclose(FILE *stream);
```
**Ascribing types to `fopen` and `fclose` (2)**

```plaintext
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) ->
[1:addr] ' (fopen_v l | ptr l)

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

*To Memory Safety through Proofs – p.18/36*
Can we also handle `fcloseall`?

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

\[
\begin{align*}
\text{sorts} & \quad \sigma ::= b \mid \sigma_1 \rightarrow \sigma_2 \\
\text{csorts} & \quad \sigma_c ::= (\sigma_1, \ldots, \sigma_n) \Rightarrow \sigma \\
\text{sta. terms} & \quad s ::= a \mid sc(s_1, \ldots, s_n) \mid \lambda a : \sigma.s \mid s_1(s_2) \\
\text{sta. var. ctx.} & \quad \Sigma ::= \emptyset \mid \Sigma, a : \sigma
\end{align*}
\]

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

\[
\begin{align*}
1 & : \text{() } \Rightarrow \text{type} \\
\text{true} & : \text{() } \Rightarrow \text{bool} \\
\text{false} & : \text{() } \Rightarrow \text{bool} \\
\to & : \text{(type, type) } \Rightarrow \text{type} \\
\sqsubset & : \text{(bool, type) } \Rightarrow \text{type} \\
\land & : \text{(bool, type) } \Rightarrow \text{type} \\
\leq & : \text{(type, type) } \Rightarrow \text{bool} \quad \text{(impredicative formulation)}
\end{align*}
\]

Also, for each sort \(\sigma\), we assume that the two static constructors \(\forall_\sigma\) and \(\exists_\sigma\) are assigned the sc-sort \((\sigma \to \text{type}) \Rightarrow \text{type}\).
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).
Some (unfamiliar) forms of types

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

Here is an example involving both guarded and asserting types:

$$\{a : \text{int} \mid a \geq 0\}$$

$$\text{int} \ a \to [a' : \text{int} \mid a' < 0] \text{int} \ a'$$

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

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

\[ d \ ::= \ x \ | \ dc(d_1, \ldots, d_n) \ | \]
\[ \text{lam } x.d \ | \ app(d_1, d_2) \ | \]
\[ \forall^+(v) \ | \ forall^-(d) \ | \]
\[ \forall^+(v) \ | \ forall^-(d) \ | \]
\[ \land(d) \ | \ \text{let } \land(x) = d_1 \ \text{in } d_2 \ | \]
\[ \exists(d) \ | \ \text{let } \exists(x) = d_1 \ \text{in } d_2 \]

values \[ v \ ::= \ x \ | \ dcc(v_1, \ldots, v_n) \ | \ \text{lam } x.d \ | \]
\[ \forall^+(v) \ | \ \forall^+(v) \ | \ \land(v) \ | \ \exists(v) \]

Dyn. var. ctx. \[ \Delta \ ::= \ \emptyset \ | \ \Delta, x : s \]
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:

\[
\begin{align*}
\text{nil} & : \forall a : \text{type. } \text{list}(a, 0) \\
\text{cons} & : \forall a : \text{type.} \forall n : \text{int.} \\
& \quad n \geq 0 \supset ((a, \text{list}(a, n)) \Rightarrow \text{list}(a, n + 1))
\end{align*}
\]
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:

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

```plaintext
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:

\[
\begin{align*}
0 \times n &= 0 \\
(m + 1) \times n &= m \times n + n \\
(-m) \times n &= -(m \times n)
\end{align*}
\]
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 : nat. \forall p : int. \text{MUL}(m, n, p) \to (p \geq 0) \land 1

We need to verify that \textit{lemma} is a total function:

- $\langle m \rangle$ is a termination metric.
- \textit{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.
Time for a demo

The demo is about various implementations of the factorial function ...
Related work

Here is only a fraction:

- Theorem proving systems: NuPrl, Coq, ...
- (Meta) Logical Frameworks: Twelf, ...
- Functional Languages: Cayenne, Delphin, Dependent ML, Omega, RSP1, Vera, ...
- Separation logic, ...
- Clay, Effective theory of refinements, $L^3$, Vault, ...
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.