# *Programming with C Library Functions Safely*

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 ($\mathcal{ATS}$). 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)

# *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**

# *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 meaning of the concrete syntax is given as follows:

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

# *Some function declarations in ATS (1)*

```
fun tail_safe {a:type, n:int | n > 0}
    (xs: list (a, n)): list (a, n-1) =
    // [case*]: exhaustive pattern matching
    case* xs of _ :: xs' => xs'

exception SubscriptException

fun head {a:type, n:int | n >= 0}
    (xs: list (a, n)): [n > 0] a =
  case* xs of
    | x :: _ => x
    | '[] => raise SubscriptException
```

# *Some function declarations in ATS (2)*

The types of the two previously defined functions can be formally written as follows:

$$
\begin{aligned}
tail\_safe &: \forall a : type. \forall n : int. \\
&\qquad n > 0 \supset (list(a, n) \rightarrow list(a, n-1)) \\
head &: \forall a : type. \forall n : int. \\
&\qquad n \geq 0 \supset (list(a, n) \rightarrow n > 0 \wedge a)
\end{aligned}
$$

# *An illustrative example*

Suppose that we define the following function to compute the tail of a given (possibly empty) list:

```
fun tail {a:type, n:int | n >= 0}
    (xs: list (a, n)): [n > 0] list (a, n-1) =
  let
     val _ = head (xs)
     // [n > 0] is established here!
  in
     tail_safe (xs)
  end
```

We see that $head$ acts like a proof function showing that a given list is not empty.

# *An illustrative example (contd)*

- If $head$ were a total function, namely, a function that is pure and terminating, then there would really be no need to execute the code $head(xs)$.
- Of course, $head$ is not a total function, and therefore the code $head(xs)$ cannot be erased.
- However, the need to distinguish total proof functions from program functions that may not be total is made clear in this case.

# *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.

# *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 | $\sigma_c$ | $::=$ | $(\sigma_1, \ldots . \sigma_n) \Rightarrow \sigma$ |
| sta. terms | $s$ | $::=$ | $a \mid sc(s_1, \ldots, s_n) \mid \lambda a : \sigma . s \mid s_1(s_2)$ |
| sta. var. ctx. | $\Sigma$ | $::=$ | $\emptyset \mid \Sigma, a : \sigma$ |

- 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*

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

Also, for each sort $\sigma$, 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).

# *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:

```
{a:int | a >= 0}
  int a -> [a': int | a' < 0] int a'
```

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

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

# *Syntax for dynamics*

| | | | |
|---|---|---|---|
| dyn. terms | $d$ | $::=$ | $x \mid dc(d_1, \ldots, d_n) \mid$ |
| | | | $\textbf{lam}\ x.d \mid \textbf{app}(d_1, d_2) \mid$ |
| | | | $\supset^+(v) \mid \supset^-(d) \mid$ |
| | | | $\forall^+(v) \mid \forall^-(d) \mid$ |
| | | | $\wedge(d) \mid \textbf{let}\ \wedge(x) = d_1\ \textbf{in}\ d_2 \mid$ |
| | | | $\exists(d) \mid \textbf{let}\ \exists(x) = d_1\ \textbf{in}\ d_2$ |
| values | $v$ | $::=$ | $x \mid dcc(v_1, \ldots, v_n) \mid \textbf{lam}\ x.d \mid$ |
| | | | $\supset^+(v) \mid \forall^+(v) \mid \wedge(v) \mid \exists(v)$ |
| 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 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:

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

# *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:

$$\forall m : nat. \forall n : nat. \forall p : int.\ \ \underline{\mathbf{MUL}}(m, n, p) \rightarrow (p \geq 0) \wedge \mathbf{1}$$

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.

# *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*

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

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

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

$$\textit{set\_ptr} \quad : \quad \forall a_1 : type. \forall a_2 : type. \forall l : addr.$$
$$(a_1@l \mid \mathbf{ptr}(l), a_2) \rightarrow (a_2@l \mid \mathbf{1})$$

# *The swap function*

```
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:

$$
\begin{array}{lcl}
array\_v\_none & : & \forall a : type. \forall l : addr.\ \ array\_v(a, 0, l) \\
array\_v\_some & : & \forall a : type. \forall n : nat. \forall l : addr. \\
 & & \quad (a@l, array\_v(a, n, l+1)) \rightarrow \\
 & & \quad array\_v(a, n+1, l)
\end{array}
$$

# *Viewtypes*

A viewtype is a linear type.

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

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

# *Accessing the first element of an array*

```
fun 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
```

$$get\_first \;:\; \forall a : type.\forall n : int.\forall l : addr.n > 0 \supset (array\_v(a,n,l) \mid \mathbf{ptr}(l)) \rightarrow (array\_v(a,n,l) \mid a)$$

# *A proof function for view change*

```
prfun take_out_lemma
      {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 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 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 $free\_v(I, L)$ 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, ...
- (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.