

# Programming with Theorem Proving in ATS

## (Functional Part)

Hongwei Xi

Boston University

Work partly funded by NSF grant CCR-0702665

# *Motivation for the research*

To support advanced types such as dependent types and linear types in practical programming.

- Pure type inference in the style of ML is no longer available.
- The programmer may need to construct proofs to validate type equality.

# *What is ATS?*

ATS is a programming language with a type system rooted in the framework *Applied Type System*. A variety of programming styles are currently supported in ATS, which include:

- Functional Programming
- Imperative Programming (with Explicit Pointers)
- Object-Oriented Programming
- Multithreaded Programming (with Pthreads)

More information on ATS is available at:

<http://www.ats-lang.org>

# *Current status of ATS*

The current implementation of ATS is done in Objective Caml. After type-checking, programs in ATS are compiled into C (by `atscc`) and then into assembly (by `gcc`).

# *Fibonacci function (version 1)*

```
fun fib1 (x: int): int =  
  if x > 1 then fib1 (x-1) + fib1 (x-2) else x
```

# *Fibonacci function (version 2)*

```
// [Nat] is the type for natural numbers  
typedef Nat = [i:int | i >= 0] int (i)
```

```
fun fib2 (x: Nat): Nat =  
  if x > 1 then fib2 (x-1) + fib2 (x-2) else x
```

# *Fibonacci function (version 3)*

```
fun fib3 (x: Nat): Nat =
  let
    fun loop (x: Nat, a0: Nat, a1: Nat): Nat =
      if x > 0 then loop (x-1, a1, a0 + a1)
      else a0
    in
      loop (x, 0, 1)
    end
```

# *Fibonacci function (version 4)*

```
dataprop FIB (int, int) = // specification
| FIB_bas_0 (0, 0)
| FIB_bas_1 (1, 1)
| {i:nat} {r0,r1:int}
    FIB_ind (i+2, r0+r1) of (FIB (i, r0), FIB (i+1, r1))
```

Some explanation of the syntax:

$FIB\_bas\_0$  :  $() \rightarrow FIB(0, 0)$

$FIB\_bas\_1$  :  $() \rightarrow FIB(1, 1)$

$FIB\_ind$  :  $\forall i : nat. \forall r_0 : nat. \forall r_1 : nat.$

$(\mathbf{FIB}(i, r_0), \mathbf{FIB}(i + 1, r_1)) \rightarrow \mathbf{FIB}(i + 2, r_0 + r_1)$

# *Fibonacci function (version 4)*

```
fun fib4 {n:nat} .<n>.
  (x: int n): [r:int] (FIB (n, r) | int r) = // implementation
  let
    fun loop {i,j:nat | i+j == n} {r0,r1:int}
      (pf0: FIB (j, r0), pf1: FIB (j+1, r1) |
       x: int i, a0: int r0, a1: int r1)
      : [r:int] (FIB (n, r) | int r) =
      if x > 1 then
        loop (pf1, FIB_ind (pf0, pf1) | x-1, a1, a0 + a1)
      else (pf0 | a0)
  in
    loop (FIB_bas_0 (), FIB_bas_1 () | x, 0, 1)
  end
```

*fib4* :  $\forall n : \text{nat}. \text{int}(n) \rightarrow \exists r : \text{int}. (\mathbf{FIB}(n, r) \mid \text{int}(r))$

*loop* :  $\forall i : \text{nat}. \forall j : \text{nat}. \forall r_0 : \text{int}. \forall r_1 : \text{int}. (i + j = n) \supset$   
 $(\mathbf{FIB}(j, r_0), \mathbf{FIB}(j + 1, r_1) \mid \text{int}(i), \text{int}(r_0), \text{int}(r_1)) \rightarrow$   
 $\exists r : \text{int}. (\mathbf{FIB}(n, r) \mid \text{int}(r))$

# *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:
  - a static component (statics), where types are formed and reasoned about, and
  - a 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 ( $\lambda C$ )

# *Type equality*

- The notion of type equality plays a pivotal rôle in type system design. However, the importance of this role is often less evident in commonly studied type systems. For instance,
  - The simply typed  $\lambda$ -calculus: two types are considered equal if and only if they are syntactically the same;
  - The second-order polymorphic  $\lambda$ -calculus: two types are considered equal if and only if they are  $\alpha$ -equivalent;
  - The higher-order polymorphic  $\lambda$ -calculus: two types are considered equal if and only if they are  $\beta\eta$ -equivalent.
- The situation immediately changes when dependent types come into the picture.

# *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*

**1** :  $() \Rightarrow type$

*true* :  $() \Rightarrow bool$

*false* :  $() \Rightarrow bool$

$\rightarrow$  :  $(type, type) \Rightarrow type$

$\supset$  :  $(bool, type) \Rightarrow type$

$\wedge$  :  $(bool, type) \Rightarrow type$

$\leq$  :  $(type, type) \Rightarrow bool$  (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).

# *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 function declaration in ATS*

```
fun concat {a:type} {m,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 piece of 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.

# *A dataprop declaration in ATS*

```
dataprop MUL (int, int, int) =  
  | {n:int} MULbas (0, n, 0)  
  | {m,n,p:int | m >= 0}  
    MULind (m+1, n, p+n) of MUL (m, n, p)  
  | {m,n,p:int | m > 0}  
    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,n:nat} {p:int} .<m>.
  (pf: MUL (m, n, p)): [p >= 0] void =
  case+ pf of
    | MULbas () => ()
    | MULind pf' =>
      let prval _ = lemma pf' in () end
```

The proof function proves:

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

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

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

# *An example of PwTP*

```
fun concat {a:type} {m,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.

# *Related work*

Here is only a fraction:

- Theorem proving systems: NuPrl, Coq, ...
- (Meta) Logical Frameworks: Twelf, ...
- Functional Languages: Delphin, Omega, ...
- Dependently Typed Functional Languages: Cayenne, Dependent ML, Epigram, ...

# *Conclusion*

- We have outlined a design to support programming with theorem proving.
- In addition, we have carried out this design in the programming language ATS.
- This is a flexible design. In addition to intuitionistic proofs, linear proofs can also be constructed and manipulated in ATS to support reasoning on resources such as memory. Please find more details about ATS at:

<http://www.ats-lang.org>

# *The end of the talk*

**Thank you!**