Combining Programming with Theorem Proving

Chiyan Chen  and  Hongwei Xi

Boston University
**Motivation for the Research**

- To support advanced type systems for practical programming, where
  - pure type inference (as is supported in ML) is no longer available, and
  - the programmer may have to construct proofs to validate type equality.
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.
ATS

ATS is a programming language with a type system rooted in the framework \textit{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)
- Meta-programming (via backquote/comma notation) (available)
- Assembly programming (under development)
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 20k 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:

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

fun head {a:type, n:int | n >= 0} (xs: list (a, n)): [n > 0] a =
  case* xs of
    | x :: _ => x
    | '[] => raise Subscript
Some function declarations in ATS (2)

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

\[
\begin{align*}
tail\_safe & : \forall a : \text{type}. \forall n : \text{int}. \\
& \quad \quad n > 0 \supset (\text{list}(a, n) \rightarrow \text{list}(a, n - 1)) \\
head & : \forall a : \text{type}. \forall n : \text{int}. \\
& \quad \quad n \geq 0 \supset (\text{list}(a, n) \rightarrow n > 0 \land \text{list}(a, n - 1))
\end{align*}
\]
An illustrative example

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

```plaintext
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 \( \text{head} \) were a total function, namely, a function that is pure and terminating, then there would really be no need to execute the code \( \text{head}(xs) \).

- Of course, \( \text{head} \) is not a total function, and therefore the code \( \text{head}(xs) \) 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:

- 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 ($\lambda C$)
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 $bool$.

\[
sorts \quad \sigma ::= b \ | \ \sigma_1 \to \sigma_2 \\
c sorts \quad \sigma_c ::= (\sigma_1, \ldots, \sigma_n) \Rightarrow \sigma \\
sta. \ terms \quad s ::= a \ | \ sc(s_1, \ldots, s_n) \ | \ \lambda a : \sigma. s \ | \ s_1(s_2) \\
sta. \ var. \ ctx. \quad \Sigma ::= \emptyset \ | \ \Sigma, a : \sigma
\]

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 : () ⇒ type
true : () ⇒ bool
false : () ⇒ bool
→ : (type, type) ⇒ type
⊑ : (bool, type) ⇒ type
∧ : (bool, type) ⇒ type
≤ : (type, type) ⇒ 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 $(\sigma → type) ⇒ type$. 
**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: $B \land T$
- Guarded type: $B \supset T$

Here is an example involving both guarded and asserting types:

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

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

**dyn. terms**

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

**values**

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

**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: 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 piece of code currently cannot pass type-checking in ATS because non-linear constraints on integers are involved.
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: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:

\[
0 \times n = 0
\]
\[
(m + 1) \times n = m \times n + n
\]
\[
(-m) \times n = -(m \times n)
\]
A proof function declaration in ATS

```plaintext
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:

\[
\forall m : nat. \forall n : nat. \forall p : int. \text{MUL}(m, n, p) \rightarrow (p \geq 0) \land 1
\]

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 programming with theorem proving

```plaintext
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.
```
Related work

Here is only a fraction:

- Theorem proving systems: NuPrl, Coq, ...
- (Meta) Logical Frameworks: Twelf, ...
- Functional Languages: Delphin, Omega, Vera, ...
- Dependently Typed Functional Languages: Cayenne, Dependent ML, Epigram, RSP1, ...
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.

This is highly flexible design and we are currently also keen to formally support reasoning on properties such as memory allocation/deallocation, deadlocks, race conditions, etc.

Along such directions, linear proofs (in addition to intuitionistic proofs) are to be constructed and manipulated. More details about ATS can be found at: http://www.cs.bu.edu/~hwxi/ATS
The end of the talk

Thank you!