Implementing an evaluator for mini-ML in ATS: a case of programming with theorem proving

Hongwei Xi

Boston University

Work partly funded by NSF grant CCR-0229480
Programming with theorem proving

- What is it?
- Where do we need it?
- How can we support it?
- This is fundamentally different from the paradigm of extracting programs from proofs.
- There are many design issues. Here is an analogy:

\[ E = mc^2 \]  \quad \longrightarrow \quad \textit{a nuclear power plant}
**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.
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 $\textit{bool}$ and $\textit{type}$.

sorts $\sigma ::= b \mid \sigma_1 \rightarrow \sigma_2$
c-sorts $\sigma_c ::= (\sigma_1, \ldots, \sigma_n) \Rightarrow \sigma$
sta. terms $s ::= a \mid \textit{sc}(s_1, \ldots, s_n) \mid \lambda : \sigma.s \mid s_1(s_2)$
sta. var. ctx. $\Sigma ::= \emptyset \mid \Sigma, a : \sigma$

Let us use $B$ and $T$ for static terms of sorts $\textit{bool}$ and $\textit{type}$, respectively.
Some static constants

\[
\begin{align*}
1 & : (\) \Rightarrow \text{type} \\
\top & : (\) \Rightarrow \text{bool} \\
\bot & : (\) \Rightarrow \text{bool} \\
\rightarrow & : (\text{type}, \text{type}) \Rightarrow \text{type} \\
\exists & : (\text{bool}, \text{type}) \Rightarrow \text{type} \\
\land & : (\text{bool}, \text{type}) \Rightarrow \text{type} \\
\leq_{tp} & : (\text{type}, \text{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 \( (\sigma \rightarrow \text{type}) \Rightarrow \text{type} \).
Some (unfamiliar) forms of types

- Guarded type: $B ⊇ T$
- Asserting type: $B ∧ T$

Here is an example involving both guarded and asserting types:

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

where $\text{int}(I)$ is a singleton type for the integer equal to $I$. This type can be assigned to a function from nonnegative integers to negative integers.
A constraint relation is of the following form:

$$\Sigma; \vec{B} \models B$$

where $\vec{B}$ stands for a (possibly empty) sequence of static boolean terms (often referred to as assumptions).

Here is an interesting question:

- Is deduction modulo a special case where $\vec{B}$ is empty?
Syntax for dynamics

\[
\begin{align*}
\text{dyn. terms} & \quad d ::= x \mid dc(d_1, \ldots, d_n) \mid \\
& \quad \text{lam } x.d \mid \text{app}(d_1, d_2) \mid \\
& \quad \exists^+(v) \mid \exists^-(d) \mid \\
& \quad \forall^+(v) \mid \forall^-(d) \mid \\
& \quad \land(d) \mid \text{let } \land(x) = d_1 \text{ in } d_2 \mid \\
& \quad \exists(d) \mid \text{let } \exists(x) = d_1 \text{ in } d_2 \\
\text{values} & \quad v ::= x \mid dc(v_1, \ldots, v_n) \mid \text{lam } x.d \mid \\
& \quad \exists^+(v) \mid \forall^+(v) \mid \land(v) \mid \exists(v) \\
\text{dyn. var. ctx.} & \quad \Delta ::= \emptyset \mid \Delta, x : s
\end{align*}
\]
Typing judgment

A typing judgment is of the following form:

$$\Sigma; \vec{B}; \Delta \vdash d : T$$
ATS

ATS is a programming language with a type system rooted in the framework $\mathcal{ATS}$. In ATS, a variety of programming paradigms are supported in a typeful manner, including:

- Functional programming (available)
- Object-oriented programming (available)
- Imperative programming with pointers (available)
- Modular programming (available)
- Assembly programming (under development)

Here is the homepage of ATS:

http://www.cs.bu.edu/~hwxi/ATS/ATS.html
The concrete syntax means the following:

\[
\begin{align*}
\text{nil} & : \forall a : \text{type. list}(a, 0) \\
\text{cons} & : \forall a : \text{type}. \forall n : \text{int.} \\
& \quad n \geq 0 \supset ((a, \text{list}(a, n)) \Rightarrow \text{list}(a, n + 1))
\end{align*}
\]
A datatype declaration in ATS (2)

```plaintext
datatype list (type, int) =
  | {a:type} nil (a, 0)
  | {a:type, n:nat}
      cons (a, n+1) of (a, list (a, n))
```

The concrete syntax means the following:

\[
\begin{align*}
\text{nil} & : \forall a : \text{type}. \ list(a, 0) \\
\text{cons} & : \forall a : \text{type}. \forall n : \text{nat}. \ (a, list(a, n)) \Rightarrow list(a, n + 1)
\end{align*}
\]
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 \textit{append} is assigned the following type:

\[
\forall a : \text{type}. \forall m : \text{nat}. \forall n : \text{nat}.
(l\text{ist}(a, m), l\text{ist}(a, n)) \rightarrow l\text{ist}(a, m + n)
\]
fun concat {a: type, m: nat, n: nat} (xss: list (list (a, n), m)) : 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.
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

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:

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

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

\( \langle m \rangle \) is a termination metric.

\( \text{case}\star \) 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.
Implementing mini-ML in ATS (1)

types

\[ T ::= \text{nat} | T_1 \times T_2 | T_1 \rightarrow T_2 \]

expressions

\[ e ::= x | z | s(e) | \langle e_1, e_2 \rangle | \text{fst}(e) | \text{snd}(e) | \lambda x.e | \text{app}(e_1, e_2) | \text{let } x = e_1 \text{ in } e_2 | \text{case } e \text{ of } (z \Rightarrow e_1 | s(x) \Rightarrow e_2) | \text{fix } x.e \]

values

\[ v ::= x | z | s(v) | \langle v_1, v_2 \rangle | \lambda x.e \]

Both the static semantics and the dynamic semantics of mini-ML can be defined in a standard manner. Let us write \( e \rightarrow v \) to mean that the expression \( e \) evaluates to the value \( v \) in mini-ML.
Implementing mini-ML in ATS (2)

- We are to form a representation \( \text{rep}(e) \) for expressions \( e \) in mini-ML.
- We are to form a representation \( \text{rep}(v) \) for values \( v \) in mini-ML.
- We are to implement a function \( \text{evaluate} \) such that if the functional call \( \text{evaluate}(\text{rep}(e)) \) returns, then it returns \( \text{rep}(v) \) satisfying \( e \rightarrow v \).

We now use the pure call-by-value \( \lambda \)-calculus in the place of mini-ML so as to make the presentation more accessible.
A first-order representation for pure $\lambda$-terms

datatype EXP =
    | One   | Shi of EXP
    | Lam of EXP | App of (EXP, EXP)

The variables in $\lambda$-terms are represented as deBruijn indexes. For instance, $\lambda x.\lambda y.y(x)$ can be represented as

$$\text{Lam} (\text{Lam} (\text{App} (\text{One}, \text{Shi} (\text{One}))))$$

It is straightforward to prove the adequacy of the representation.
A higher-order representation for pure $\lambda$-terms

datatype EXP =
  Lam of (EXP -> EXP) | App of (EXP, EXP)

The variables in $\lambda$-terms are represented as meta-variables, that is, variables in the meta-language. For instance, $\lambda x.\lambda y.y(x)$ can be represented as

$$\text{Lam}(\text{lam } x \Rightarrow \text{Lam}(\text{lam } y \Rightarrow \text{App}(y, x)))$$

In the dynamics of ATS, this is not an adequate representation. In particular, there are many values of type EXP that do not correspond to any (pure) $\lambda$-terms.
Representing $\lambda$-terms in statics

datasort exp =
   | lm of (exp -> exp) | ap of (exp, exp)

datasort exps = none | more of (exps, exp)

Note that this is an *adequate* representation in the statics of ATS.
Representing λ-terms in dynamics

datatype IN (exp, exps) =
  | {es:exp, e:exp} One (more (es, e), e)
  | {es:exp, e1:exp, e2:exp} Shi (more (es, e2), e1) of EXP (es, e1)

datatype EXP (exp, exps) =
  | {es:exp, e:exp} Var (es, e) of IN (e, es)
  | {es:exp, f:exp -> exp} Lam (es, lm f) of
    | {e:exp} EXP (more (es, e), f e)
  | {es:exp, e1:exp, e2:exp} App (es, ap (e1, e2)) of
    | (EXP (es, e1), EXP (es, e2))
Intuitively, we use a value of type \( \text{EXP}(e_1 :: e_2 :: \ldots :: e_n, e) \) to represent the expression \( e \), where the (at most \( n \)) variables in the value represents \( e_1, \ldots, e_n \). For instance,

\[ \lambda x, \lambda y. y(x) \] is represented as

\[ \text{Lam} (\text{Lam} (\text{App} (\text{Var \ INone}, \ \text{Var\ (INshi \ INone)}))) \]
Specifying call-by-value evaluation strategy

\[
\begin{array}{c}
\lambda x.e \rightarrow \lambda x.e \\
\hline
\end{array}
\]

\[
\begin{array}{c}
e_1 \rightarrow \lambda x.e \\
\hline
\end{array}
\]

\[
\begin{array}{c}
e_2 \rightarrow v_1 \\
\hline
\end{array}
\]

\[
\begin{array}{c}
e[x := v_1] \rightarrow v_2 \\
\hline
\end{array}
\]

\[
\begin{array}{c}
e_1(e_2) \rightarrow v_2 \\
\hline
\end{array}
\]

\[
\text{dataprop EVAL (exp, exp) =}
\]

\[
\begin{array}{c}
| \{f: \text{exp} \rightarrow \text{exp}\} \text{ EVALlam of (lm f, lm f)}
\end{array}
\]

\[
\begin{array}{c}
| \{e_1: \text{exp}, e_2: \text{exp}, f: \text{exp} \rightarrow \text{exp}, v_1: \text{exp}, v_2: \text{exp}\}
\end{array}
\]

\[
\begin{array}{c}
\text{EVALapp (ap (e_1, e_2), v_2) of}
\end{array}
\]

\[
\begin{array}{c}
(\text{EVAL(e_1, lm f), EVAL(e_2, v_1), EVAL(f v_1,v_2))}
\end{array}
\]
Specifying call-by-name evaluation strategy

\[
\begin{align*}
\lambda x. e & \rightarrow \lambda x. e \quad \text{(lam)} \\
\quad\quad e_1 & \rightarrow \lambda x. e \\
\quad\quad e[x := e_2] & \rightarrow e_3 \quad \text{(app)}
\end{align*}
\]

\[
\text{dataprop EVAL (exp, exp) =} \\
\quad \{ f: \text{exp} \rightarrow \text{exp} \} \ \text{EVALlam of (lm f, lm f)} \\
\quad \{ e_1: \text{exp}, e_2: \text{exp}, f: \text{exp} \rightarrow \text{exp}, e_3: \text{exp} \} \\
\quad \quad \text{EVALapp (ap (e_1, e_2), e_3) of} \\
\quad \quad \quad (\text{EVAL(e_1, lm f), EVAL(f e_2, e_3))}
\]

Implementing an evaluator for mini-ML in ATS: a case of programming with theorem proving – p.28/35
Representing values

dataprop ISVAL (exp) =
    | {f:exp -> exp} ISVALlam (lm f)

datatype VAL (exp) =
    | {vs:exps, f:exp -> exp}
        VALclo (lm f) of // closure
            (ENV vs, {v:exp} EXP (more (vs, v), f v))

and ENV (exps) = // environment
    | ENVnone (none)
    | {vs:exps, v:exp}
        ENVmore (more (vs, v)) of
            (ISVAL v | ENV vs, VAL v)
Some proof functions

If $v$ is a value, then $v \rightarrow v$ holds:

\[
\text{prfun lemma1} \{v: \text{exp}\} .< >.
\]
\[
\begin{array}{l}
\text{(pf: ISVAL v): EVAL (v, v) = }
\text{case}^* \text{ pf of ISVALlam} \Rightarrow \text{EVALlam}
\end{array}
\]

If $e \rightarrow v$ holds, then $v$ is a value:

\[
\text{prfun lemma2} \{e: \text{exp}, v: \text{exp}\} .<??>.
\]
\[
\begin{array}{l}
\text{(pf: EVAL (e, v): ISVAL v = }
\text{case}^* \text{ pf of}
\begin{array}{l}
\text{EVALlam} \Rightarrow \text{ISVALlam}
\text{EVALapp (_, _, pf3) \Rightarrow lemma2 pf3}
\end{array}
\end{array}
\]
Implementing evaluation

fun eval {vs:exps, e:exp} (env: ENV (vs), e: EXP (vs, e)) : [v:exp] '(EVAL (e, v) | VAL v) =
case e of
  | Var i => evalVar (env, i)
  | Lam body => '(EVALlam | VALclo (env, body))
  | EXPapp (e1, e2) =>
    let
      val '(pf1 | VALclo (env', body)) = eval (env, e1)
      val '(pf2 | arg) = eval (env, e2)
      val '(pf3 | v) =
        eval (ENVmore (lemma2 pf2 | env', arg), body)
    in
      '(EVALapp (pf1, pf2, pf3) | v)
    end

and evalVar {vs:exps, v:exp} (env: ENV vs, i: IN (v, vs)) : '(EVAL (v, v) | VAL v) = ...
fun evaluate {e:exp} (e: EXP (none, e)):
  [v:exp] '(EVAL (e, v) | VAL v) =
  eval (ENVnil, e)

So formally, the function evaluate is assigned the following type:

\[ \forall e : exp. EXP(none, e) \rightarrow \exists v : exp. EVAL(e, v) * VAL(v) \]
Related work

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

Implementing an evaluator for mini-ML in ATS: a case of programming with theorem proving – p.33/35
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.

The theme of programming with theorem proving is much broader than what is given in this talk. For instance, ATS is also equipped with a form of linear logic (similar to but different from separation logic) to reason about memory properties, which forms the basis for supporting safe use of pointers (e.g., pointer arithmetic).

We are currently also keen to formally support reasoning on properties such as deadlocks and race conditions.
The end of the talk

Thank you!