Combining Higher-Order Abstract Syntax with First-Order Abstract Syntax in ATS

Kevin Donnelly, Hongwei Xi

Boston University
Combining HOAS with FOAS

Goals: To use HOAS to encode syntax and semantics, while allowing

1. verified meta-programming as partial recursive functions, and
2. encoding of meta-theory as total recursive functions.

Outline:

- Brief overview of ATS
  - types, props, termination and coverage
- Encoding object syntax and semantics
  - Combines HOAS and FOAS
- Verification of programs on object language
  - Specification over HOAS, programming over FOAS
## ATS: Statics and Dynamics

<table>
<thead>
<tr>
<th></th>
<th>Pure</th>
<th>Effectful</th>
</tr>
</thead>
<tbody>
<tr>
<td>statics (pure,</td>
<td>Simply-sorted lambda calculus with distinguished sorts <em>type</em> and <em>prop</em> which act as types for dynamics</td>
<td>Disallowed</td>
</tr>
<tr>
<td>(pure, decidable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>equality)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dynamics</td>
<td>Well-founded recursion and exhaustive case analysis (assigned props)</td>
<td>General recursion, exceptions, references (assigned types)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Statics includes integers and arithmetic, with equality checked by a decision procedure.
Proofs are total programs, this is enforced by the type checker:

- Recursion must be well founded - guaranteed by checking a programmer supplied metric

- All cases must be covered - the decision procedure must be able to prove that unlisted cases are impossible (i.e. they imply $\perp$)

- Dataprops may not contain negative occurrences, unlike datatypes
Representing Object Language

We will use the untyped lambda calculus as an example

\[ M ::= x \mid \lambda x. M \mid M_1(M_2) \]

We use the following general approach to encoding object languages.

- Syntax: represented in the statics where there is no recursion
- Semantics: judgments as types (or props), deductions as terms
- Meta-theory: directly encoded as total recursive functions
- Meta-programming: partial recursive functions
Representing Object Language

The static representation is standard.

\[ \text{ap} : (tm, tm) \rightarrow tm \]
\[ \text{lm} : (tm \rightarrow tm) \rightarrow tm \]

\[ \text{\(\Gamma x \Downarrow\)} = x \]
\[ \text{\(\Gamma \lambda x. M \Downarrow\)} = \text{lm(\(\lambda x : tm. \ \Gamma M \Downarrow\))} \]
\[ \text{\(\Gamma M_1(M_2) \Downarrow\)} = \text{ap(\(\Gamma M_1 \Downarrow\), \(\Gamma M_2 \Downarrow\))} \]

Since the statics is simply typed lambda calculus with products and constants, this representation is adequate.
Representation in Dynamics

A sort for $tm$ lists to serve as a context for deBruijn terms:
static sort $tms$ is a list of $tm$ terms:

$$
\text{nil} : tms \quad \text{cons} : (tm, tms) \rightarrow tms
$$

Datatype $\text{IN}(t : tm, ts : tms)$ represents a context lookup $t \in ts$. Isomorphic to natural numbers, serves as deBruijn indices.

$\text{INone} : \forall ts : tms. \forall t : tm. \text{IN}(t, t :: ts)$

$$
\text{INone} \quad \frac{t \in t :: ts}{\text{INone}}
$$

$\text{INshi} : \forall ts : tms. \forall t : tm. \forall t' : tm. \text{IN}(t, ts) \rightarrow \text{IN}(t, t' :: ts)$

$$
\text{INshi} \quad \frac{t \in ts}{\text{INshi}}
\quad \frac{t \in t' :: ts}{\text{INshi}}
$$

Combining HOAS and FOAS in ATS – p.7
Representation in Dynamics

The datatype \( \text{TERM}(ts : tms, t : tm, n : int) \) represents a term in context, \( ts \vdash_n t \), integer index is size.

\[
\begin{align*}
\text{VAR} : & \forall ts : tms. \forall t : tm. \text{IN}(t, ts) \rightarrow \text{TERM}(ts, t, 0) \\
\frac{t \in ts}{ts \vdash_0 t} & \quad \text{VAR} \\
\text{LAM} : & \forall ts : tms. \forall f : tm \rightarrow tm. \forall n : \text{nat.} \\
& \quad (\forall x : tm. \text{TERM}(x :: ts, f(x), n)) \\
& \quad \rightarrow \text{TERM}(ts, lm(f), n + 1) \\
\frac{x, ts \vdash_n f(x)}{ts \vdash_{n+1} lm(f)} & \quad \text{LAM} \\
\text{APP} : & \forall ts : tms. \forall t_1 : tm. \forall t_2 : tm. \forall n_1 : \text{nat.} \forall n_2 : \text{nat.} \\
& \quad (\text{TERM}(ts, t_1, n_1), \text{TERM}(ts, t_2, n_2)) \\
& \quad \rightarrow \text{TERM}(ts, \text{ap}(t_1, t_2), n_1 + n_2 + 1) \\
\frac{ts \vdash_{n_1} t_1 \quad ts \vdash_{n_2} t_2}{ts \vdash_{n_1+n_2+1} \text{ap}(t_1, t_2)} & \quad \text{APP}
\end{align*}
\]

\( \forall x_1 : tm \ldots \forall x_n : tm. \text{TERM}(x_1 :: \ldots :: x_n :: \text{nil}, t, m) \) represents \( x_1, \ldots, x_n \vdash_m t \)

Ground terms in the context are explicit substitutions.

For adequacy, need to use the fact that universal quantification over \( tm \) is parametric.
We abbreviate \( \exists n : \text{nat. TERM}(t, ts, n) \) by \( \text{TERM}0(t, ts) \).

Examples:

\[
\begin{align*}
\text{TERM}0(nil, \lambda x. \lambda y. y(x)) & \quad \{ \text{LAM(LAM(APP(0, 1)))} \} \\
\forall x : tm. \forall y : tm. \text{TERM}0(x :: y :: nil, \lambda z. y) & \quad \{ \text{LAM(1)} \} \\
\text{TERM}0(\lambda x. x :: \lambda x. x :: nil, \lambda x. x) & \quad \{ 0, 1, \text{LAM}(0) \}
\end{align*}
\]
Implementing Substitution

We must code substitution on dynamic terms as a function

\[ subst : \forall t : tm. \forall ts : tms. \forall t' : tm. \forall n : nat. \]
\[ (\text{TERM}(t :: ts, t', n), \text{TERM0}(ts, t)) \rightarrow \text{TERM0}(ts, t') \]
We use the judgments-as-types principle.

Big-step call-by-value semantics as a prop, \( \text{EVAL}(tm, tm, int) \). The last argument counts constructors.

\[
\begin{align*}
\text{EVALlam} & : \ \forall f : tm \to tm. \text{EVAL}(lm f, lm f, 0) \\
\text{EVALapp} & : \ \forall t_1 : tm. \forall t_2 : tm. \forall f_1 : tm \to tm. \forall v_2 : tm. \forall v : tm. \forall n_1 : nat. \\
& \quad \forall n_2 : nat. \forall n_3 : nat. \\
& \quad \text{EVAL}(t_1, lm f_1, n_1), \text{EVAL}(t_2, v_2, n_2), \text{EVAL}(f_1 v_2, v, n_3)) \\
& \quad \to \text{EVAL}(ap(t_1, t_2), v, n_1 + n_2 + n_3 + 1)
\end{align*}
\]
Evaluation implemented with environment and closures, verified with respect to simpler eval rules.

We begin by declaring (mutually inductive) datatypes to represent values ($\text{VAL}(tm)$) and environments ($\text{ENV}(tms)$).

\[
\begin{align*}
\text{VALclo} & : \ \forall ts : tms. \forall f : tm \rightarrow tm. (\text{ENV}(ts), \text{TERM}_0(ts, lm f)) \rightarrow \text{VAL}(lm f) \\
\text{ENVnil} & : \ \text{ENV}(\text{nil}) \\
\text{ENVcons} & : \ \forall ts : tms. \forall t : tm. (\text{ENV}(ts), \text{VAL}(t)) \rightarrow \text{ENV}(t :: ts)
\end{align*}
\]
We can now implement the evaluation function:

\[
\text{eval} : \forall ts : \text{tms}. \forall t : \text{tm}. \forall p : \text{nat}. \\
\quad (\text{TERM}(ts, t), \text{ENV}(ts)) \\
\quad \rightarrow \exists v : \text{tm}. (\text{EVAL0}(t, v) \mid \text{VAL}(v))
\]

The type of the function guarantees its partial correctness. After erasing props (which is done after type-checking) we have a function with type:

\[
\forall ts : \text{tms}. \forall t : \text{tm}. \forall p : \text{nat}. \\
\quad (\text{TERM}(ts, t), \text{ENV}(ts)) \\
\quad \rightarrow \exists v : \text{tm}. (\text{VAL}(v))
\]

which is the type we expect for an eval function.
We have used this encoding technique to implement several other, more complicated, examples:

- Verified evaluator for (monomorphic) Mini-ML with references (ref cells specified as list, implemented as array).
- Proved call-by-value normalization (as a total recursive function) for simply typed $\lambda$-calculus using Tait’s method.
Related and Future Work

Most closely related work:

- Miller and McDowell’s FO$\lambda^\Delta$ supports a similar representation
- Ambler, Crole and Momigliano’s Hybrid uses a similar representation but in Isabelle

Future possibilities:

- Add higher-order pattern matching so we can encode reduction under lambda.
- Add locally-scoped constants (a la Miller and Tiu) for more flexible treatment of variables (to encode, e.g., transitivity of algorithmic subtyping of $F^\leq$)

ATS implementation and this example along with other available at:

http://www.cs.bu.edu/~hwxi/ATS/ATS.html