Static name control for FreshML

François Pottier

LICS
July 13th, 2007

INRIA

Here is an archetypical FreshML algebraic data type definition:

$$
\begin{aligned}
&\textbf{type}\ term = \\
&\quad \mid Var\ \textbf{of}\ atom \\
&\quad \mid Abs\ \textbf{of}\ \langle atom \rangle term \\
&\quad \mid App\ \textbf{of}\ term \times term
\end{aligned}
$$

In short, *FreshML* [Pitts and Gabbay, 2000] extends ML with primitive expression- and type-level constructs for *atoms* and *abstractions*.

What is the point?

This allows transformations to be defined in a natural style:

```
fun sub accepts a, t, s =
  case s of
  | Var (b) →
      if a = b then t else Var (b)
  | Abs (b, u) →
      Abs (b, sub(a, t, u))
  | App (u, v) →
      App (sub (a, t, u), sub (a, t, v))
  end
```

The dynamic semantics of FreshML dictates that, in the *Abs* case, the atom *b* is automatically chosen *fresh* for both *a* and *t*. The term *u* is renamed accordingly. As a result, *no capture* can occur.

Properties and non-properties of FreshML

Shinwell and Pitts [2005] have shown that abstractions cannot be violated: that is, an abstraction effectively *hides the identity* of its bound atom.

Unfortunately, *not every FreshML function denotes a mathematical function*, because fresh name generation is a computational effect.

For instance, here is a flawed code snippet:

```
fun eta_reduce accepts t =
  case t of
  | Abs (x, App (e, Var (y))) →
      if x = y then e else ...
  | ...
```

A slogan

Ideally, a FreshML compiler should check that every function is pure. This requires ensuring that *freshly generated atoms do not escape*, or, in other words, that they are eventually bound.

Paraphrasing an epigram by Perlis, the compiler should ensure that

> *there is (in the end) no such thing as a free atom!*

This would not just make the language prettier — it would help catch bugs.

Towards domain-specific program proof

Just like type-checking, the task is in principle easy, but overwhelming for a human. It is a prime candidate for *automation*.

It is, however, slightly more ambitious than traditional type-checking. We are looking at a kind of *domain-specific program proof*.

Manual specifications (preconditions, postconditions, etc.) will sometimes be required, but all proofs will be fully automated.

My contribution is to:

- introduce a *simple logic* for reasoning about values and sets of atoms, equipped with a (slightly conservative) *decision procedure*;
- allow logical assertions to serve as *preconditions* and *postconditions* and to appear *within* algebraic data type definitions;
- exploit *alphaCaml*'s flexible language [Pottier, 2006] for defining algebraic data types with binding structure.

Where proof obligations arise

Generating a fresh atom x for use in an expression e produces:

- a *hypothesis* that x is fresh for all pre-existing objects;
- a *proof obligation* that x is fresh for the result of e.

(Two objects o_1 and o_2 are *fresh for* one another when they have disjoint *support*, that is, disjoint sets of free atoms. This is written $o_1 \# o_2$.)

Here is an excerpt of the capture-avoiding substitution function:

```
fun sub accepts a, t, s =
  case s of
  | Abs (b, u) →
      Abs (b, sub(a, t, u))
  | ...
```

Matching against *Abs* yields the hypothesis $b \# a, t, s$ and the proof obligation $b \# Abs(b, sub(a, t, u))$ — a tautology, since b is never in the support of $Abs(b, \ldots)$.

A more subtle example

Here is an excerpt of a "β_0-reduction" function for λ-terms:

```
fun reduce accepts t =
  case t of
  | App (Abs (x, u), Var (y)) →
      reduce (sub (x, Var (y), u))
  | ...
```

Proving that x is not in the support of the value produced by the right-hand side requires *some knowledge about the semantics* of capture-avoiding substitution.

Providing a postcondition

This knowledge is provided via an explicit *postcondition*:

$\textbf{fun}\ \mathit{sub}\ \textbf{accepts}\ a,\ t,\ s$
$\textbf{produces}\ u\ \textbf{where}\ \textbf{free}(u) \subseteq \textbf{free}(t) \cup (\textbf{free}(s) \setminus \textbf{free}(a)) =$
$\quad \ldots$

This produces a *new hypothesis* within *reduce* and *new proof obligations* within *sub*.

First, the benefit:

```
fun reduce accepts t =
  case t of
  | App (Abs (x, u), Var (y)) →
      reduce (sub (x, Var (y), u))
  | ...
```

The postcondition for *sub*, together with the hypothesis that x is fresh for y, tells us that *x is fresh for sub(x, Var(y), u)*.

Furthermore, by (recursive) assumption, *reduce is pure and has empty support*, so x is fresh for the entire right-hand side, as desired.

Then, the obligations:

$$
\begin{array}{l}
\textbf{fun}\ sub\ \textbf{accepts}\ a,\ t,\ s \\
\textbf{produces}\ u\ \textbf{where}\ \textbf{free}(u) \subseteq \textbf{free}(t) \cup (\textbf{free}(s) \setminus \textbf{free}(a)) = \\
\quad \textbf{case}\ s\ \textbf{of} \\
\quad \mid\ Var\ (b) \rightarrow \\
\qquad\quad \textbf{if}\ a = b\ \textbf{then}\ t\ \textbf{else}\ Var\ (b) \\
\quad \mid\ \ldots
\end{array}
$$

The postcondition is *propagated down* into each branch of the case and if constructs and *instantiated* where a value is returned. For instance, inside the Var/else branch, one must prove

$$free(Var(b)) \subseteq free(t) \cup free(s) \setminus free(a)$$

At the same time, branches give rise to *new hypotheses*. Inside the Var/else branch, we have $s = Var(b)$ and $a \neq b$.

How do we check that

$$\left.\begin{array}{r} s = Var(b) \\ a \neq b \end{array}\right\} \quad \text{imply} \quad \mathsf{free}(Var(b)) \subseteq \mathsf{free}(t) \cup \mathsf{free}(s) \setminus \mathsf{free}(a) \quad ?$$

Well, $s = Var(b)$ implies $\mathsf{free}(s) = \mathsf{free}(Var(b))$ by *congruence*, and $\mathsf{free}(Var(b))$ is $\mathsf{free}(b)$ by *definition*.

Furthermore, since a and b have type *atom*, $a \neq b$ is equivalent to $\mathsf{free}(a) \# \mathsf{free}(b)$.

There remains to check that

$$\left.\begin{array}{r}\mathrm{free}(s) = \mathrm{free}(b)\\ \mathrm{free}(a) \mathrel{\#} \mathrm{free}(b)\end{array}\right\} \quad \text{imply} \quad \mathrm{free}(b) \subseteq \mathrm{free}(t) \cup \mathrm{free}(s) \setminus \mathrm{free}(a)$$

No knowledge of the semantics of free is required to prove this, so let us replace free(a) with A, free(b) with B, and so on...

(A, B, S, T denote finite sets of atoms.)

There remains to check that

$$\left.\begin{array}{r} S = B \\ A \mathbin{\#} B \end{array}\right\} \quad \text{imply} \quad B \subseteq T \cup S \setminus A$$

This is initially an assertion about finite *sets of atoms*, but one can prove that its truth value is unaffected if we interpret it in a *2-point* Boolean algebra:

$$\left.\begin{array}{r} (\neg S \vee B) \wedge (\neg B \vee S) \\ \neg(A \wedge B) \end{array}\right\} \quad \text{imply} \quad \neg B \vee T \vee (S \wedge \neg A)$$

So, the decision problem reduces to SAT.

(The reduction is incomplete. See the paper for the fine print!)

As a slightly more advanced example, here are excerpts of a version of *normalization by evaluation* of untyped λ-terms.

The algorithm is essentially a *closure-based interpreter* for possibly open terms, combined with a decompiler.

Source terms are just λ-terms.

```
type term =
  | TVar of atom
  | TLam of ⟨ atom × inner term ⟩
  | TApp of term × term
```

Nothing new, except I now use *alphaCaml* syntax: in *TLam*(x, t), the atom x is bound within the term t.

Semantic values and environments

Semantic values are very much like source terms, except λ-abstractions carry an explicit *environment*.

$$
\begin{array}{l}
\textbf{type}\ \mathit{value} = \\
\quad \mid \mathit{VVar}\ \textbf{of}\ \mathit{atom} \\
\quad \mid \mathit{VClosure}\ \textbf{of}\ \langle\ \mathit{env} \times \mathit{atom} \times \textbf{inner}\ \mathit{term}\ \rangle \\
\quad \mid \mathit{VApp}\ \textbf{of}\ \mathit{value} \times \mathit{value}
\end{array}
$$

$$
\begin{array}{l}
\textbf{type}\ \mathit{env}\ \textbf{binds} = \\
\quad \mid \mathit{ENil} \\
\quad \mid \mathit{ECons}\ \textbf{of}\ \mathit{env} \times \mathit{atom} \times \textbf{outer}\ \mathit{value}
\end{array}
$$

In *VClosure(env, x, t)*, the atoms in the domain of *env*, written ***bound***(*env*), as well as the atom *x*, are bound within the term *t*.

Evaluation of a term t under an environment *env* produces a value v, whose support is predicted by an *explicit postcondition*.

fun *evaluate* **accepts** *env*, *t* **produces** *v*
where **free**(v) $\subseteq$ **outer**(*env*) $\cup$ (**free**(t) $\setminus$ **bound**(*env*))

(Code omitted.)

Decompilation (reification) translates a semantic value back to a source term.

```
fun decompile accepts v produces t
= case v of
   | VVar (x) →
       TVar (x)
   | VClosure (cenv, x, t) →
       TLam (x, decompile (evaluate (cenv, t)))
   | VApp (v1, v2) →
       TApp (decompile (v1), decompile (v2))
  end
```

In the closure case, the body is evaluated, without introducing an explicit binding for x, so that x remains a symbolic name. *evaluate's postcondition* guarantees that the atoms in the domain of *cenv* do not escape.

Last, *normalization* is the composition of evaluation and decompilation.

fun *normalize* **accepts** *t* **produces** *u*
= *decompile* (*evaluate* (*ENil* , *t*))

The system accepts these definitions, which guarantees that *normalize* denotes a *mathematical function* of terms to (⊥ or) terms.

Summary

During this talk, I have argued in favor of semi-automated, *static name control* for FreshML.

A toy implementation exists and has been used to prove the correctness of a few standard code manipulation algorithms, involving flat *environments*, nested *contexts*, nested *patterns*, etc.

See the paper (and its extended version) for details, examples, and a comparison with related work.

Future work

In the future, I would like to:

- *extend* the current toy implementation with first-class functions, mutable state, exceptions, extra primitive operations, etc.;
- *combine* the decision procedure with a general-purpose automated first-order theorem prover.

I would like to see a version of (Fresh)ML where programs are decorated with assertions expressed in a general-purpose logic, so as to guarantee not only that atoms are properly bound, but also that programs are *correct*.

Pitts, A. M. and Gabbay, M. J. 2000.
A metalanguage for programming with bound names modulo renaming.
In *International Conference on Mathematics of Program Construction (MPC)*. Lecture Notes in Computer Science, vol. 1837. Springer Verlag, 230–255.

Pottier, F. 2006.
An overview of Caml.
In *ACM Workshop on ML*. Electronic Notes in Theoretical Computer Science, vol. 148. 27–52.

Shinwell, M. R. and Pitts, A. M. 2005.
On a monadic semantics for freshness.
Theoretical Computer Science 342, 28–55.