Static name control for FreshML

François Pottier

May 28th, 2007

INRIA

Everyone in this room has seen some variant of this archetypical FreshML type definition:

$$
\begin{array}{l}
\textbf{\textit{type}}\ \mathit{term} = \\
\quad \mid\ \mathit{Var}\ \textbf{\textit{of}}\ \mathit{atom} \\
\quad \mid\ \mathit{Abs}\ \textbf{\textit{of}}\ \langle \mathit{atom} \rangle \mathit{term} \\
\quad \mid\ \mathit{App}\ \textbf{\textit{of}}\ \mathit{term} \times \mathit{term}
\end{array}
$$

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

What is the point?

The point is to allow transformations to be defined in a natural style:

```
1 fun sub accepts a, t, s =
2   case s of
3   | Var (b) →
4       if a = b then t else Var (b)
5   | Abs (b, u) →
6       Abs (b, sub(a, t, u))
7   | App (u, v) →
8       App (sub (a, t, u), sub (a, t, v))
9   end
```

The dynamic semantics of FreshML dictates that, on line 5, the name b is automatically chosen *fresh* for both a and t. The term u is renamed accordingly. As a result, *no capture* can occur.

FreshML abstractions are opaque

Shinwell and Pitts [2005] have shown that the encodings of two alpha-equivalent terms are observationally equivalent.

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.

Can you spot the flaw in this code snippet?

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

Ideally, a FreshML compiler should check that *freshly generated names do not escape*, or, in other words, that every function is pure.

Paraphrasing a famous quote — thanks, Dale — the compiler should ensure that

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

The required check is exactly the same as in Berghofer and Urban's nominal package.

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 names, 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 for defining algebraic data types with binding structure.

Where proof obligations arise

Wherever we write **fresh** x **in** e, we get:

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

An analogous phenomenon takes place when matching against an abstraction pattern.

This is the well-known *freshness condition for binders*.

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:

$$
\begin{array}{l}
\textbf{fun}\ \mathit{reduce}\ \textbf{accepts}\ t = \\
\quad \textbf{case}\ t\ \textbf{of} \\
\quad \mid \mathit{App}\ (\mathit{Abs}\ (x,\ u),\ \mathit{Var}\ (y)) \longrightarrow \\
\qquad\quad \mathit{reduce}\ (\mathit{sub}\ (x,\ \mathit{Var}\ (y),\ u)) \\
\quad \mid\ \ldots
\end{array}
$$

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*:

$\textbf{fun}$ 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*.

free denotes the free atoms (or support) function. It is defined at every type.

First, the benefit:

fun *reduce* **accepts** t =
  case t **of**
  | *App* (*Abs* (x, u), *Var* (y)) $\longrightarrow$
    reduce (*sub* (x, *Var* (y), u))
  | ...

The postcondition for *sub*, together with the (free) 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:

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

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

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

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

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

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

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

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 the 2-point algebra of *Booleans:*

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

So, the decision problem reduces to SAT.

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

Outline

Normalization by evaluation

As a slightly more advanced example, here is a version of *normalization by evaluation* of untyped λ-terms in *50 lines* of code.

Source terms

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*.

```
type value =
  | VVar of atom
  | VClosure of ⟨ env × atom × inner term ⟩
  | VApp of value × value

type env binds =
  | ENil
  | ECons of env × atom × outer value
```

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

The keyword ***binds*** means that the type *env* is intended to appear within abstraction brackets $\langle \cdot \rangle$.

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) ⊆ outer(env) ∪ (free(t) \ bound(env))
= case t of
  | TVar (x) →
      case env of
      | ENil →
          VVar (x)
      | ECons (tail, y, v) →
          if x = y then v else evaluate (tail, t) end
      end
```

When t is a variable, the environment is looked up, in a straightforward way.

(continued on next slide)

Evaluation 2

When t is a λ-abstraction, a *closure* is constructed.

```
  | TLam (x, t) →
      VClosure (env, x, t)
```

The binding structure of this closure is such that, in this case, *evaluate's postcondition is trivially satisfied!*

(continued on next slide)

Evaluation 3

When t is an application, each side is reduced in turn. If a β-redex appears, it is reduced by evaluating the closure's body *under an appropriate environment*.

```
27   | TApp (t1, t2) →
28       let v1 = evaluate (env, t1) in
29       let v2 = evaluate (env, t2) in
30       case v1 of
31       | VClosure (cenv, x, t) →
32           evaluate (ECons (cenv, x, v2), t)
33       | v1 →
34           VApp (v1, v2)
35       end
36   end
```

Note that, on line 32, writing *env* instead of *cenv*, or failing to create a binding for *x*, *would cause the code to be rejected*, even though it would still be type-correct!

Decompilation

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 λ-abstraction 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 names in the domain of *cenv* do not escape.

Normalization

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

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

The system accepts these definitions: *normalize* denotes a *mathematical function* of terms to ($\perp$ 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 [Pottier, 2007] 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.

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. 2007.
Static name control for FreshML.
In *IEEE Symposium on Logic in Computer Science (LICS)*. To appear.

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