Static name control for FreshML

François Pottier

What does ML stand for?

ML is supposed to be a *Meta-Language*...

... so it must be good at manipulating abstract syntax, right?

Why ML is inadequate

Here is an ML algebraic data type for λ-terms:

$$
\begin{array}{l}
\mathsf{type}\ \mathit{term} = \\
\quad \mid\ \mathit{Var}\ \mathsf{of}\ \mathit{string} \\
\quad \mid\ \mathit{Abs}\ \mathsf{of}\ \mathit{string} \times \mathit{term} \\
\quad \mid\ \mathit{App}\ \mathsf{of}\ \mathit{term} \times \mathit{term} \\
\quad \mid\ \mathit{Let}\ \mathsf{of}\ \mathit{string} \times \mathit{term} \times \mathit{term}
\end{array}
$$

Now, try formulating *capture-avoiding substitution*, for instance... The task will be *heavy* and *error-prone*.

The problem is, ML deals with *sums* and *products*, but does not know about *binders*.

Representing the λ-calculus in FreshML

To remedy this shortcoming, *FreshML* (Pitts & Gabbay, 2000) makes *names* and *binding* (also known as *atoms* and *abstractions*) primitive notions.

Here is a FreshML algebraic data type for λ-terms:

$$
\begin{array}{l}
\mathbf{type}\ \mathit{term} = \\
\quad \mid\ \mathit{Var}\ \mathbf{of}\ \mathit{atom} \\
\quad \mid\ \mathit{Abs}\ \mathbf{of}\ \langle\ \mathit{atom} \times \mathbf{inner}\ \mathit{term}\ \rangle \\
\quad \mid\ \mathit{App}\ \mathbf{of}\ \mathit{term} \times \mathit{term} \\
\quad \mid\ \mathit{Let}\ \mathbf{of}\ \langle\ \mathit{atom} \times \mathbf{outer}\ \mathit{term} \times \mathbf{inner}\ \mathit{term}\ \rangle
\end{array}
$$

Now, capture-avoiding substitution *can* be written in a natural way...

Example: capture-avoiding substitution

```
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    | Let (x, u1, u2) →
10       Let (x, sub (a, t, u1), sub (a, t, u2))
11   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.

Why (unrestricted) FreshML is inadequate

So far, so good. But FreshML allows defining *bizarre* “functions”:

```
fun bv accepts x =
  case x of
  | Var (a) →
      empty
  | Abs (a, t) →
      singleton(a) ∪ bv(t)
  | App (t, u) →
      bv(t) ∪ bv(u)
  | ...
```

The dynamic semantics of FreshML dictates that, for a fixed term t, every call to $bv(t)$ returns a (distinct) set of *fresh* atoms!

Why (unrestricted) FreshML is inadequate

By letting freshly generated names *escape* their scope, FreshML allows defining “functions” whose semantics is not a mathematical function — that is, *impure* functions.

But nobody would write code like *bv*, right?

Why (unrestricted) FreshML is inadequate

Can you spot the flaw in this more subtle example?

$$
\begin{array}{l}
\textsf{fun } \mathit{optimize} \textsf{ accepts } t = \\
\quad \textsf{case } t \textsf{ of} \\
\quad \mid \mathit{Abs}\ (x, \mathit{App}\ (e, \mathit{Var}\ (y))) \rightarrow \\
\qquad \textsf{if } x = y \textsf{ then } \mathit{optimize}\ (e) \textsf{ else next case} \\
\quad \mid \ldots
\end{array}
$$

Ideally, a FreshML compiler should check that names do not escape — which also means that all functions are *pure*. In short, we need *static name control* for FreshML.

Towards domain-specific program proof

Isn't that too ambitious? Shouldn't this issue be left aside until someone comes by and *proves* the program correct?

Proofs about names are easy in principle, but also easy to drown in. This means that they are prime candidates for *full automation*.

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 automatic.

State of the art

Pitts and Gabbay's "FreshML 2000" did have static name control, enforced via a type system that could keep track of, and establish, freshness assertions.

This type system was abandoned circa 2003, because it was too limited.

Sheard and Taha's MetaML avoids the problem by tying name generation and name abstraction together, at a significant cost in expressiveness.

Contribution

My contribution is to:

- introduce a *rich logic* for reasoning about values and sets of names, together with a *conservative decision procedure* for this logic;
- allow logical assertions to serve as function *preconditions* or *postconditions* and to appear *inside* algebraic data type definitions;
- exploit C*a*ml's flexible language for defining algebraic data types with binding structure.

What do we prove?

What does it mean for an atom *not to escape* its scope?

What requirements should we impose on the code?

How do we know that these requirements are sufficient to ensure that valid programs have *pure* meaning?

The answer is in *nominal set theory* (Gabbay & Pitts, 2002).

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.

A simple example

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 $b \# Abs(b, sub(a, t, u))$, which is easily discharged, since b is never in the support of $Abs(b, \ldots)$.

A more subtle example

Here is an excerpt of an "optimization" function for λ-terms:

```
fun optimize accepts t =
  case t of
  | Let (x, Var (y), u) →
      optimize (sub (x, Var (y), u))
  | ...
```

How do we prove that x does not appear in the support of the value produced by the right-hand side? We need *precise knowledge* of the behavior of capture-avoiding substitution.

Assertions

Let us add to our definition of capture-avoiding substitution (already shown) an explicit *postcondition*:

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

This has a double effect: produce a *new hypothesis* inside “optimize” and *new proof obligations* inside “sub”.

Benefits inside "optimize"

```
fun optimize accepts t =
  case t of
  | Let (x, Var (y), u) →
      optimize (sub (x, Var (y), u))
  | ...
```

The postcondition for "sub" tells us that

$$x \text{ is fresh for } sub(x, Var(y), u),$$

which implies that

$$x \text{ is also fresh for } optimize(sub(x, Var(y), u)).$$

Indeed, in Pure FreshML, *functions cannot make up new (free) names!*

Obligations inside "sub"

```
fun sub accepts a, t, s
produces u where free(u) ⊆ free(t) ∪ (free(s) \ free(a)) =
  case s of
  | Var (b) →
      if a = b then t else Var (b)
  | ...
```

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 **else** branch, one must prove

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

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

Discharging proof obligations

How do we check that

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

Well, $s = \mathit{Var}(b)$ implies $\mathit{free}(s) = \mathit{free}(\mathit{Var}(b))$ by *congruence*, and $\mathit{free}(\mathit{Var}(b))$ is $\mathit{free}(b)$ by *definition* of the type "term".

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

Discharging proof obligations

There remains to check that

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

No knowledge about the semantics of *free* is required to prove this, so let us replace $\mathit{free}(a)$ with A, $\mathit{free}(b)$ with B, and so on...

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

Discharging proof obligations

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 it turns out that its truth value is unaffected if we view it as an assertion about *Booleans*:

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

Think of this shift of perspective as *focusing on a single atom*.

Discharging proof obligations

Finally, the assertion boils down to the *unsatisfiability* of

$$(\neg S \vee B) \wedge (\neg B \vee S) \wedge (\neg A \vee \neg B) \wedge B \wedge \neg T \wedge (\neg S \vee A)$$

which a SAT solver will prove fairly easily (an understatement).

Reducing all proof obligations down to Boolean formulæ obviates the need for a set of *ad hoc* proof rules.

The reduction is *incomplete*, but comes “reasonably close” to completeness...

One source of incompleteness

Replacing every set expression of the form $free(x)$ with a set variable X is always *sound* — if we can prove that the property holds of an arbitrary set X, then also holds of the particular set $free(x)$.

It is *complete* only if $free(x)$ *can actually* denote every possible set of atoms.

However, because *the type of x is known*, this is not necessarily the case.

One source of incompleteness

For instance, if x has integer type, then $free(x)$ denotes the empty set. If x has type **atom**, then $free(x)$ denotes a singleton set. And so on...

To mitigate this source of incompleteness, I translate $free(x)$ to:

- $\emptyset$, when *every inhabitant* of the type of x has empty support;
- X, together with the constraint $X \neq \emptyset$, when *no inhabitant* of the type of x has empty support;
- X, as before, otherwise.

The logic allows stating $X = \emptyset$ and $X \neq \emptyset$, but does not allow further reasoning about cardinality.

The full constraint language, as of today

$$
\begin{array}{lcll}
s & ::= & \mathit{free}(v) \mid \emptyset \mid \mathbb{A} \mid s \cap s \mid s \cup s \mid \neg s & \text{set expressions} \\
F & ::= & b \mid 0 \mid 1 \mid F \wedge F \mid F \vee F \mid \neg F & \text{Boolean expressions} \\
C & ::= & F \Rightarrow s = \emptyset \mid s \neq \emptyset \mid v = v \mid C \wedge C & \text{constraints}
\end{array}
$$

Here, v ranges over values of arbitrary type, while b ranges over variables of type "bool".

Normalization by *evaluation*

This was put forward by Shinwell, Pitts and Gabbay (2003) as a piece of code whose well-behavedness is difficult to establish.

It is *accepted* by Pure FreshML up to three changes:

- replacing first-class functions with *explicit data structures*;
- decorating these data structures with appropriate *binding information*;
- annotating the main function with a *postcondition*.

(The absence of first-class functions may be a temporary limitation.)

Conversion to *A*-normal form

This transformation simplifies complex, tree-structured expressions by *hoisting out* and *naming* intermediate computations. It is defined as follows:

Evaluation contexts:

$$E ::= [] \mid \textbf{let } x = E \textbf{ in } e \mid E\ e \mid v\ E \mid \ldots$$

Transformation rules (freshness side-conditions implicit!):

$$\begin{aligned} E[\textbf{let } x = e_1 \textbf{ in } e_2] &\longrightarrow \textbf{let } x = e_1 \textbf{ in } E[e_2] \\ E[v_1\ v_2] &\longrightarrow \textbf{let } x = v_1\ v_2 \textbf{ in } E[x] \end{aligned}$$

Conversion to *A*-normal form

I know of two ways of implementing this transformation:

- Flanagan *et al.*'s *continuation-passing style* algorithm, in the style of Danvy and Filinski; for people who write two-level programs in their sleep...
- a direct-style, *context-passing style* algorithm; for mere mortals.

Perhaps surprisingly, Flanagan *et al.*'s algorithm is *easily proved correct* in Pure FreshML (modulo defunctionalization).

```
(define normalize-term (lambda (M) (normalize M (lambda (x) x))))

(define normalize
  (lambda (M k)
    (match M
      [`(lambda ,params ,body) (k `(lambda ,params ,(normalize-term body)))]
      [`(let (,x ,M1) ,M2) (normalize M1 (lambda (N1) `(let (,x ,N1) ,(normalize M2 k))))]
      [`(if0 ,M1 ,M2 ,M3) (normalize-name M1 (lambda (t) (k `(if0 ,t ,(normalize-term M2) ,(normalize-term M3)))))]
      [`(,Fn . ,M*) (if (PrimOp? Fn)
                        (normalize-name* M* (lambda (t*) (k `(,Fn . ,t*))))
                        (normalize-name Fn (lambda (t) (normalize-name* M* (lambda (t*) (k `(,t . ,t*)))))))]
      [V (k V)])))

(define normalize-name
  (lambda (M k)
    (normalize M (lambda (N) (if (Value? N) (k N) (let ([t (newvar)]) `(let (,t ,N) ,(k t))))))))

(define normalize-name*
  (lambda (M* k)
    (if (null? M*)
        (k '())
        (normalize-name (car M*) (lambda (t) (normalize-name* (cdr M*) (lambda (t*) (k `(,t . ,t*)))))))))
```

Conversion to *A*-normal form

I wrote another algorithm, which avoids continuations and manipulates explicit *contexts* — terms with a hole.

The algorithm's main function, *split*, accepts a term t and produces a pair of a context C and a term u such that t has the same meaning as $C[u]$.

The code is straightforward, but coming up with an adequate type definition for contexts was not immediate.

Floating up contexts

The contexts that are floated up are defined by:

$$C ::= [] \mid \mathsf{let}\ x = e\ \mathsf{in}\ C$$

So, when a context of the form

$$\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = e_n\ \mathsf{in}\ []$$

is eventually filled with an expression e,

- occurrences of x_i in e become bound;
- occurrences of x_i in $e_{i+1}, \ldots, e_n$ become bound.
- occurrences of x_i in $e_1, \ldots, e_i$ *remain free*.

η-expansion, fixed

The corrected code is *accepted*:

```
fun optimize accepts t =
  case t of
  | Abs (x, App (e, Var (y))) →
      if x = y and not member (x, free(e))
      then optimize (e)
      else next case
  | ...
```

Note that =, *and*, *not*, *member*, and *free* are simply *primitive operations* with accurate specifications — and *if* is just syntactic sugar for *case* over Booleans.

Some primitive operations

Here are the specifications for these built-in functions:

$(=)$ accepts x, y produces b where $b \rightarrow \mathsf{free}(x) = \mathsf{free}(y)$
where not $b \rightarrow \mathsf{free}(x) \# \mathsf{free}(y)$

and accepts x, y produces z where $z = (x$ and $y)$

not accepts x produces y where $y =$ not x

member accepts x, s produces b where $b \rightarrow \mathsf{free}(x) \subseteq \mathsf{free}(s)$
where not $b \rightarrow \mathsf{free}(x) \# \mathsf{free}(s)$

free accepts x produces s where $\mathsf{free}(s) = \mathsf{free}(x)$

A to-do list

There remains a wealth of ideas to explore in order to turn Pure FreshML into a realistic meta-programming language:

- local functions;
- mutable state;
- exceptions;
- extra primitive operations;
- multiple sorts of atoms;
- type & sort polymorphism, parameterized algebraic data types;
- non-linear patterns;
- safe non-freshening.

Safe non-freshening

Sometimes, it is *safe* to match against an abstraction *without freshening* its bound atoms:

```
let t = ... in
case ... of
| Abs (x, u) →
    Abs (x, App (u, u))      // freshening not required
| Abs (x, u) →
    Abs (x, App (t, u))      // freshening required
| Abs (x, u) →
    sub (x, t, u)            // freshening not required
```

But *when* is it safe and *how* do we prove it?

Nominal sets

Atoms are drawn from a countably infinite set $\mathbb{A}$.

A *nominal set* X is equipped with an action of the finite permutations of atoms on the elements of X such that every element has finite support.

The *support* of an element $x \in X$ is the least set of atoms outside of which no permutation affects x.

Types as nominal sets

Every type of FreshML will be interpreted as a nominal set, which effectively means that the operations of *renaming* and *support* are available at all types.

Nominal sets are typically constructed out of other nominal sets via a combination of the following constructions:

$\mathbb{A}$	the universe of atoms
$X_1 \times X_2$	product
$X_1 + X_2$	sum
$\langle \mathbb{A} \rangle X$	the *abstractions* over elements of X
$X_1 \rightarrow X_2$	the *finitely supported* functions of X_1 into X_2
$\mu(F)$	least fixed point

Freshness

Two elements x_1, x_2 are *fresh for* one another iff x_1 and x_2 have disjoint support. This is written $x_1 \# x_2$.

A property P is said to be true of *some/any sufficiently fresh atom* a if and only if P holds of all but a finite set of atoms. This is written $\mathit{NEW}\ a.P$.

Locally fresh atoms

Key fact. Let f be an element of the nominal function space $\mathbb{A} \longrightarrow X$ such that

$$\mathsf{NEW}\ a. \quad a \# f(a) \qquad \text{— } f \text{ does not leak } a$$

That is, for some/any sufficiently fresh atom a, the image of a through f does not have a in its support. Then, there exists a unique element x of X such that

$$\mathsf{NEW}\ a. \quad x = f(a) \qquad \begin{array}{l} \text{— } f(a) \text{ does not depend upon the choice of } a \\ \text{— provided } a \text{ is chosen sufficiently fresh} \end{array}$$

The element x is written *new* a *in* $f(a)$.

Locally fresh atoms: example

For instance, if b is a fixed atom and f maps a to $\langle a \rangle\,(b, a)$, then $a \# f(a)$ holds for all atoms a.

This means that there exists a unique x such that

$$\mathit{NEW}\ a. \quad x = \langle a \rangle\,(b, a)$$

(In fact, this holds for all atoms a except b.)

This element x is usually written *new a in* $\langle a \rangle\,(b, a)$.

Note that *new* binds the *meta-variable* a, while $\langle a \rangle$ abstracts the *atom* denoted by a.

A pure semantics for FreshML

The key fact leads directly to a denotational semantics for FreshML's *fresh* construct:

$$[\![\textsf{fresh}\ x\ \textsf{in}\ e]\!]^\bullet_\eta = \mathit{new}\ a\ \mathit{in}\ [\![e]\!]^\bullet_{\eta[x\mapsto a]}$$

Of course, this makes sense only if the key fact's requirement is met:

$$\mathit{NEW}\ a.\quad a\ \#\ [\![e]\!]^\bullet_{\eta[x\mapsto a]}$$

If we enforce this condition, then $[\![\textsf{fresh}\ x\ \textsf{in}\ e]\!]^\bullet_\eta$ is well-defined, and uniquely defined — this denotational semantics is *pure*.

This gives precise meaning to the condition *"x does not escape its scope"* and explains why it guarantees a pure semantics.