FreshML: programming with binders made easy

Murdoch J. Gabbay, 1 June 2003

Cambridge University, UK,
www.cl.cam.ac.uk/˜mjg1003

FreshML is a programming language which extends ML with constructs to facilitate metaprogramming on syntax.

FreshML emerges from FM techniques, presented in my thesis in 2001, based on an original idea of Andrew Pitts': "what if names are concrete object-level atoms". Thus natural numbers $n \in \mathbb{N}$ do not qualify, because they are not 'atomic enough'. `unit_ref` does, and we shall return to that.

We realised this idea mathematically as "FM set theory", and along with Peter White and Mark Shinwell implemented a programming language with intended semantics in FM.

FreshML

The first version of FreshML [metpbn] matched its design criteria but was unwieldy to program in. We have since improved it tremendously in this respect, and developed FreshML-Lite, described in [frepbm], a paper by the same name as this talk.

I shall talk about FreshML-Lite, but I shall call it FreshML. The language is in a state of flux as improvements are identified and incorprated into the language. Code you see here may not parse tomorrow, it might not even parse as I write.

FreshML may be obtained from `www.freshml.org`. Good for fast prototyping, implementation, theory, and sheer fun.

FreshML allows you to declare Bindable Types:

```
bindable_type name;
```

Consistent with the original FM idea, elements of bindable type have no internal structure. They are very similar to elements of `unit_ref`; we can generate them dynamically and test them for equality:

```
val a = let fresh a:name in a;
val b = let fresh b:name in b;
a=a;                    true.
a=b;                    false.
```

returns `true`, then `false`, as indicated.

Given any type `ty` and `exp:ty`, we can swap names `a:name` and `b:name` in `exp`:

```
swap a,b in a;          b.
swap b,a in a=a;        true.
val c = let fresh c:name in c;
swap c,a in
 (fn x:name => if x=a then <b,c>
                      else <c,a>);

  fn x:name => if x=b then <b,a>
                      else <a,c>.
```

We can swap polymorphically over all types, even function types, as shown. This is how FreshML differs from ML using `unit_ref`, where `f:ty1=>ty2` has no intensional properties.

Binding

Given a type ty we can form <Names>ty, “bind Names in ty”.

The type-former is

n:names, exp:ty $\longmapsto$ <n>exp:<Names>ty.

So think of <n>exp as the pair (n,exp).

The type-destructor, in pattern-matching style, is

```
let ty_abs = <n'>exp' in exp'',
```

where n' and exp' are bound.

Operationally `let <n>exp = <n'>exp' in exp''` evolves as follows: a fresh `n'` is generated, and `swap n,n' in exp''` evaluated.

For example,

```
let <a>a = <n'>m' in n';
      n.
let <a>b = <n'>m' in m';
      b.
```

where two `n` are generated fresh, one for each expression (but only one escapes into the environment).

In the underlying representation `<a>exp` *is* just `(a,exp)`, but it behaves like "`exp` with `a` bound" because whenever we destruct the expression, `a` comes out freshened to `n`.

Implementors: in the dynamics the swapping is left 'delayed' on top of `exp`. If we ever try going into the structure of `exp`, the swapping is lazily pushed down. So this is a relatively cheap operation. Of course there's plenty of room for optimisations, especially when we unpack stacks of abstractions, work on the underlying `exp`, then repackage. Abstraction by non-atomic types is a recent development to help with this. See [frepbm].

Binding

Informal correctness theorem: expressions of type `<Name>ty` up to contextual equivalence are in bijection with expressions of type `ty`, with an atom bound. For example

```
<a>a  =  <b>b
<a>b  =  <c>b
<a>(fn x:name => if x=b then <b,a>
                        else <a,c>)
=
<q>(fn x:name => if x=b then <b,q>
                        else <q,c>)
```

Thus a little magic takes place in FreshML: names `a,b,c` behave like constants (which we can generate at will with `fresh`—unlike variable symbols `x,y,z` which are fixed in the program), and yet thanks to swapping we can still bind them.

```
bindable_type Name           (* names *)
;
datatype Lambda =            (* Lambda-terms *)
    Var of Name              (* a *)
  | App of Lambda*Lambda     (* t1 t2 *)
  | Lam of <Name>Lambda      (* lam a t *)
;

val rec subst : Name*Lambda*Lambda -> Lambda =
 fn (n,Var x,s) =>
          if n=x then Var x else s
  | (n,App t1 t2,s) =>
          subst(n,t1,s) subst(n,t2,s)
  | (n,Lam <a>t,s) =>
          Lam <a>(subst(n,t,s))
;
```

Quick prototyping: λ-calculus in one slide (slide II)

. . . then we can implement whatever reduction strategy we prefer:

```
val rec cbv : Lambda -> Lambda =
 fn (App t1 t2) =>
       let val t1' = cbv t1;
           val t2' = cbv t2;
       in match t1' with Lam(<n>t1'') => subst(n,t1'',t2')
                       | t1'          => App t1' t2'
   | t => t
;

val rec cbn : Lambda -> Lambda =
 fn (App t1 t2) =>
       let val t1' = cbn t1;
       in match t1' with Lam(<n>t1'') => subst(n,t1'',t2)
                       | t1'          => App t1' t2
   | t => t
;
```

Features of FreshML

1. Explicit bindable types of names.
2. Explicit names, which behave like constants.
3. Swapping.
4. Name-abstraction `<Name>ty`. This specifies binding in the datatype declaration, as `Lam` above.
5. (Freshening) pattern-matching on name-abstractions.

Correctness theorem: Expressions of `Lam` in FreshML up to contextual equivalence are in bijection with λ-calculus terms up α-equivalence.

This proved formally in [frepbm], and we can easily see from the proof how to extend it to more general datatypes. I will talk on Friday about what was, a year ago, a novel application to the π-calculus. I conclude this talk with an extended example in more traditional vein:

Extended example (from [frepbm])

Represent expressions of a small fragment of ML with the following forms:

fn x => e	function abstraction
e_1 e_2	function application
let val x = e_1 in e_2 end	local value
let fun f x = e_1 in e_2 end	local recursive function

as follows:

```
bindable_type name
datatype expr = Vid of name
              | Fn of <name>expr
              | App of expr * expr
              | Let of expr * <name>expr
              | Letfun of
                  <name>((<name>expr) * expr)
```

Extended example

```
fun subst x e (Vid y) =
      if x # y then Vid y else e
   | subst x e (Fn (<y>e1)) =
      Fn (<y>(subst x e e1))
   | subst x e (App (e1,e2)) =
      App (subst x e e1,subst x e e2)
   | subst x e (Let (e1,<y>e2)) =
      Let (subst x e e1,<y>(subst x e e2))
   | subst x e (Letfun (<f>(<y>e1,e2))) =
      Letfun (<f>(<y>(subst x e e1),subst x e e2))
```

Extended example

```
fun remove(<x>[]) = []
  | remove(<x>(y::ys)) =
      if x # y then y::(remove(<x>ys))
               else remove(<x>ys);
fun fv(Var x) = [x]
  | fv(Lam(<x>t)) = remove(<x>(fv t))
  | fv(App(t1,t2)) = (fv t1)@(fv t2);
fun is_closed t = ((fv t)=[])
```

Meta-theorem

Swapping $(a\ b)$ commutes with first-order logic:

$$\Phi((a\ b)\cdot x_1, \ldots, (a\ b)\cdot x_n) \iff \Phi(x_1, \ldots, x_n).$$

This is Equivariance of FM set theory. For example, $(a\ b)\cdot x = (a\ b)\cdot y \iff x = y$, because $(a\ b)$ is bijective on names.

Suppose you have some program exp which satisfies $\Phi(exp)$. Then so does $(a\ b)\cdot exp$.

Corollary: FreshML correctness theorem.

Final slide

This area of research is really flourishing at the moment. Download FreshML from `www.freshml.org`. Apologies though: FreshML is evolving so rapidly that at the time of writing at least, the documentation is out-of-date.

See my talk on Friday for more FM (recounted in a more theoretical dialect).

Features of FreshML (again)

1. Explicit bindable types of names.
2. Explicit names, which behave like constants.
3. Swapping.
4. Name-abstraction `<Name>ty`. This specifies binding in the datatype declaration, as `Lam` and `expr` above.
5. (Freshening) pattern-matching on name-abstractions.