FM and the π-calculus

Murdoch J. Gabbay, March 18, 2003

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

The Issue

The structure of your data should reflect the structure of your program.
So what programs can we write using this structure?

```
datatype Lam_db =                (* lambda-calculus *)
    Var of Nat                   (* x *)
  | App of Lam_db*Lam_db         (* t1 t2 *)
  | Lam of Lam_db                (* \x t *)
;
```

The Issue

Substitution:

```
val Sub : Lam_db * Nat * Lam_db -> Lam_db = fn
                              (* [s/a]t *)
    (s,a,Var a) => s
  | (s,a,Var b) => Var b
  | (s,a,App(t1,t2)) => App(Sub(s,a,t1),Sub(s,a,t2))
  | (s,a,Lam t) => Lam(Sub(raise s,a+1,t))
;
```

A bit messy, perhaps. `raise` is defined by

The Issue

```
val raise : Lam_db -> Lam_db = fn
    (Var a) => Var (a+1)
  | (App(t1,t2)) => App(raise t1,raise t2)
  | (Lam t) => Lam(raise t)
;
```

FreshML

We allow declarations of **bindable types**. A bindable type `name` has two associated operations (this is pseudo-code):

```
bindable_type Name;              (* names *)
swap : Name * Name * 'a -> 'a; (* swapping action *)
fresh : unit -> Name;            (* fresh name *)
```

`swap` takes some `(a,b,x)` and literally swaps `a` and `b` in the representation of `x`. `fresh` just chooses a fresh name (like we can do with `unit ref`). So the only difference between FreshML and ML is this swapping operation. Now watch this...

Abstraction

We can now declare a polymorphic abstraction type-former `'a -> <Name>'a` with constructor/destructor

```
val abs : Name * 'a -> <Name>'a = fn
   (a,x) => (a,x);
val conc : <Name>'a * Name -> 'a = fn
   ((a,x),b) => swap(b,a,x);
```

(packaged up as an abstract datatype, of course). Write `abs(a,x)` as `<a>x` and `conc(x_abs,c)` as `x_abs@c`.

Abstraction

We can see that `swap(a,b,swap(a,b,x))` is identical to `x`, so

$$\texttt{(<a>x)@b} = \texttt{swap(b,a,x)}$$
$$\texttt{(<a>x)@a} = \texttt{x}.$$

We can even pattern-match on abstractions, de-sugaring

```
<a>x => f(a,x)    as
x_hat => let a=fresh() in f(a,x_hat@a)
```

—in effect we guarantee that when we decompose abstractions '**names of bound variables are chosen fresh**'.

Semantics: NOM

Fix some countably infinite **set of atoms** $a, b, c, \ldots \in \mathbb{A}$. Let a **swapping** be a function $(a\ b) : \mathbb{A} \to \mathbb{A}$ defined by

$$
\begin{array}{lll}
(b\ a)a & \stackrel{\text{def}}{=} & b \\
(b\ a)b & \stackrel{\text{def}}{=} & a \\
(b\ a)n & \stackrel{\text{def}}{=} & n \qquad n \neq a, b.
\end{array}
\tag{1}
$$

Semantics: NOM

Let $\pi, \pi', \kappa \in P_{\mathbb{A}}$ be the **set of finite permutations of atoms**, thus the subset of $\mathbb{A}^{\mathbb{A}}$ inductively generated by the swappings $(a\ b)$ and **Id** the identity on $\mathbb{A}$. This is a group with unit **Id** under functional composition $\circ$.

Let the category of **Nominal Sets** have objects sets with $P_{\mathbb{A}}$ action—

(2) $\quad \forall \pi, \pi', x.\ \pi \cdot (\pi' \cdot x) = \pi \circ \pi' \cdot x \quad$ and $\quad \mathbf{Id} \cdot x = x$

the standard rules for a permutation action. Clearly $\mathbb{A}$ is the semantics for `Name` and $(a\ b)$ the semantics for `fn x => swap(a,b,x)`.

Semantics: NOM

What makes this work is **finite support**

(3) $$\forall x \in X.\, И a, b.\, (a\ b) \cdot x = x.$$

Write $\mathbb{A}^{\mathcal{S}}$ for the set of finite sets of atoms. Write $И a.\, \Phi(a)$ for $\exists S \in \mathbb{A}^{\mathcal{S}}.\, \forall a \notin S.\, \Phi(a)$. Then (3) above means

(4) $$\forall x \in X.\, \exists S \in \mathbb{A}^{\mathcal{S}}.\, \forall a, b \notin S.\, (a\ b) \cdot x = x.$$

This reflects in the semantics that anything we can build in `FreshML`, being a finite program, will only mention finitely many names, so we have a notion of 'fresh name', referring to one of the infinitely many which we have not used yet (and it doesn't matter which because if we want to change the name, we can use `swap` to do so).

Semantics: abstraction

The semantics of <Name>X is, for those interested,

$$(X^{\mathbb{A}})/{\sim} \quad \text{where} \quad f{\sim}g \overset{\text{def}}{\Leftrightarrow} И c.\, fc = gc$$

$$(\mathbb{A} \times X)/{\sim} \quad \text{where} \quad \langle a, x\rangle{\sim}\langle b, y\rangle \overset{\text{def}}{\Leftrightarrow} И c.\, (c\, a)x = (c\, b)y;$$

equivalent definitions, where maps either way are given by what we would expect from <Name>X, namely

$$f \mapsto И a.\, \langle a, f\, a\rangle$$
$$\langle a, x\rangle \mapsto \lambda b.(b\, a)x$$

(where $\sim$ takes equivalences over the choice of a in both maps).

The π-calculus

The π-calculus is full of binding, both at the level of terms and also transitions. In a series of programs `pi-ltsb-1` to `pi-ltsb-4` I explore (increasingly smart) ways of using FreshML to program terms and transitions for this calculus. We consider `pi-ltsb-3` here. The datatypes are:

```
bindable_type Name              (* bound names *)
;
datatype Proc =                 (* pi-calculus processes *)
    Par of Proc*Proc            (* (P | P') *)
  | Res of <Name>Proc           (* nu x (P) *)
  | Rep of Proc                 (* !(P) *)
  | Out of Name*Name*Proc       (* out x y.(P) *)
  | In  of Name*(<Name>Proc)    (* in x(y).(P)  *)
  | Tau of Proc                 (* tau.(P) *)
  | Ina                         (* 0 *)
;
datatype Act =
    Actt
  | Acto of Name*Name
  | Acti of Name*Name
;
type Trn = <Name>(Act*Proc)     (* results of a transition step *)
;
```

pi-ltsb-3

The three prototypical transitions are these:

```
( Out(a,b,P), <n>(Acto(a,b), P) ) : Proc* Trn
( In(a,<b>P), <b>(Acti(a,b), P) ) : Proc* Trn
( Res(<b>Out(a,b,P)), <b>(Acto(a,b), P) ) : Proc* Trn
```

This transition system is ‘deterministic’ in the sense that it never makes any arbitrary choices about fresh names, because there aren’t any (e.g. b is bound in the third transition above). I call this property **name-regularity** and can make it mathematically precise as a property of a relation $R \subseteq X \times Y$.

The code which generates the transitions is. . .

pi-ltsb-3

```
val rec trns_of : Proc -> (Trn list) =
 fn Ina           => []
  | (Tau(P))      => [promoteAbs (Actt,P)]
  | (Out(a,b,P))  => [promoteAbs (Acto(a,b),P)]
  | (In(a,<n>P))  => [<n>(Acti(a,n),P)]
    ...
  | (Res(<n>P))   => open_rule n (trns_of P)
    ...;
val open_rule_helper : Name -> Trn -> Trn option =
 fn n => fn <m>(Acto(a,b),Q) =>
    if b#n then None else
       Some (<b>( Acto(a,b) , Q ))
           | _ => None;
```

(Open)

$$\frac{P \xrightarrow{\nu m.\overline{a}b} P'}{\nu[b]P \xrightarrow{\nu b.\overline{a}b} P'}$$

```
val comm_close_1_rule_helper :
 <Name>((Act*Proc)*(Act*Proc)) -> (Trn option) =
 fn <c>((Acto(a1,b1),Q1),(Acti(a2,b2),Q2)) =>
  if a1=a2 then (
    if b1#c then
      Some (<c>(Actt,Par( Q1,rename(<b2>Q2,b1) )))
    else
      Some (<b1>(Actt,Res(<b1>(Par( Q1,rename(<b2>Q2,b1) ))) ))
                ) else None
| _ => None;
```

(Com1)
$$\frac{P_1 \stackrel{\nu c.\overline{a}b_1}{\rightarrow} Q_1 \;\; P_2 \stackrel{\nu b_2.ab_2}{\rightarrow} Q_2}{P_1 \mid P_2 \stackrel{\nu c.\tau}{\rightarrow} Q_1 \mid Q_2\{b_1/b_2\}}$$

(Close1)
$$\frac{P_1 \stackrel{\nu b.\overline{a}b}{\rightarrow} Q_1 \;\; P_2 \stackrel{\nu b.ab}{\rightarrow} Q_2}{P_1 \mid P_2 \stackrel{\nu b.\tau}{\rightarrow} \nu[b](Q_1 \mid Q_2)}$$

pi-ltsb-4

Now a little bit of faffing around; what is `pi-ltsb-4`?

```
datatype Proc =                  (* pi-calculus processes *)
    Par of Proc*Proc             (* (P | P') *)
  | Rep of (Proc NM)             (* !(nu as P) *)
  | Out of Name*Name*Proc        (* out x y.(P) *)
  | In  of Name*(<Name>Proc)     (* in x(y).(P)  *)
  | Tau of Proc                  (* tau.(P) *)
  | Ina                          (* 0 *)
;
type ProcNM = Proc NM
;
```

`NM` is the **abstraction monad**. `'a NM` is in essence `<Name list>'a`, or if you prefer $[\mathbb{A}\text{-}List]\alpha$.

pi-ltsb-4

```
(* Monad lifting function: abs >> f applies f to the abstracted value
in abs and adds abs's abstractions to the result. *)
infix >>;
val op>> : 'b NM * ('b -> 'c NM) -> 'c NM = fn
   (<l>x, f) => <l>(f x);

datatype Act =
    Actt
  | Acto of Name*Name
  | Acti of Name*Name
;
type Trn = <Name>(Act*ProcNM)  (* results of a transition step *)
;
```

For convenience I allow myself non-linear patterns (repeats of a and l in parttern below):

```
val comm_close_1_rule_helper :
   <Name>((Act*ProcNM)*(Act*ProcNM)) -> Trn option  =
   fn <c>( (Acto(a,b1),<l>q1) , (Acti(a,b2),<l>q2) ) =>
      Some <c>(Actt , <c::l> Par(q1 , rename(<b2>q2,b1)) )
    | _ => None;
```

(Com/Close1)

$$\frac{P_1 \overset{\nu c.\overline{a}b_1}{\rightarrow} [l]Q_1 \;\; P_2 \overset{\nu b_2.ab_2}{\rightarrow} [l]Q_2}{P_1 \mid P_2 \overset{\nu c.\tau}{\rightarrow} [c :: l](Q_1 \mid Q_2\{b_1/b_2\})}$$

Of course the idea is that this is, once you get used to it, ‘simpler’. I came to Lyon amongst other things to discuss with Daniel how to use a similar trick to build models of π-calculus processes à la HD-automata (Montanari et al) or $\pi\theta$-automata (Honsell et al).

Conclusions

So yes, we can use FreshML to (simply?!) program the binding of the π-calculus.