FM binding for π-calculus transitions

Murdoch J. Gabbay, June 1, 2003

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

The Issue

Metaprogramming is programming on syntax. Metaprogramming is:

- Implement a λ-calculus.
- Implement a π-calculus.
- Prove bisimilarity/correctness/Church-Rosser on the λ-calculus.
- Prove bisimilarity/correctness or build models of a π-calculus.
- More than I know...

FM techniques are an approach to metaprogramming via a new model of syntax.

The Issue

Standard model of syntax uses parse trees. Syntax has variable symbols and binding.

We seek an intuitive mathematical model of binding (and via parse trees metaprogramming) in which variable symbols are first-class objects at meta-level.

Mathematical specification of problem

Intuitive$\longmapsto$some category which looks like the category of sets. There should be a set $\mathbb{A}$ with $a, b, c, \ldots \in \mathbb{A}$ representing object-level variable symbols.

Binding: Given *any* set T there should be a set $[\mathbb{A}]T$ with arrow

(1)
$$\begin{aligned} \mathbb{A} \times T &\longrightarrow [\mathbb{A}]T \\ \langle a, t \rangle &\longmapsto [a]t \end{aligned}$$

$[a]t$ is like the $a.t$ in $\lambda a.\, t$. The above arrow tells us "think of it as $\langle a, t \rangle$", which we often do in practice of course.

Unbinding: Given a set T there should be an arrow

$$(2) \qquad \begin{aligned} ([\mathbb{A}]T) \times \mathbb{A} &\to T \\ \langle \hat{t}, a \rangle &\mapsto \hat{t}@a \end{aligned}$$

So given an abstraction $\hat{t}$ we can **concrete** it to a body $\hat{t}@a$. This tells us we can “choose a name for the bound variable name in an abstraction”.

Also, we require $([a]t)@a = t$. We shall soon see what $([a]t)@b$ should be.

Some of you may think “but we can program that up for given T”. Maybe (depends on T; function type?)—but if we can axiomatise it intensionally (over a whole category), this is roughly equivalent to delegating binding to the compiler, so we do not *have* to program it up for every given T. That is in a nutshell the story of FreshML.

Overview of FreshML

FreshML allows you to declare Bindable Types:

```
bindable_type name;
```

This is like $\mathbb{A}$. Elements are much like `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.

Binding

Given a type `ty` we can form `<Names>ty`, "bind `Names` in `ty`". This is $[\mathbb{A}]T$. The type-former is

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

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

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

See (1) and (2). We find we are only interested in opening up an abstraction at fresh `n'`. The FreshML interpreter generates `n'` fresh and evaluates `ty_abs@n'` (whatever that means). FreshML is stateful and keeps a counter of the last generated fresh name, to guarantee this.

Swapping

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.

Swapping

FreshML thus differs from ML with `unit_ref`, where `f:ty1=>ty2` has no intensional properties. We shall soon develop the mathematical model and show that we can axiomatise swapping abstractly as an intensional property of sets, which is why we dare make it polymorphic over all types in the programming language.

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

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

(3)
$$\begin{aligned} (b\ a)a &\stackrel{\text{def}}{=} b \\ (b\ a)b &\stackrel{\text{def}}{=} a \\ (b\ a)n &\stackrel{\text{def}}{=} n \qquad n \neq a, b. \end{aligned}$$

Semantics: NOM

Let $\pi, \pi', \kappa \in P_{\mathbb{A}}$ be the **set of finite permutations of atoms**, the subgroup of $\mathbb{A}^{\mathbb{A}}$ generated by $(a\ b)$ under functional composition $\circ$. The unit **Id** is $\lambda a.a$ the identity on $\mathbb{A}$.

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

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

$(a\ b)$ the semantics for `fn x => swap(a,b,x)`.

Objects have **finite support**

(5) $$\forall x \in X.\ \text{И} a, b.\ (a\ b) \cdot x = x.$$

Write $\mathcal{P}_{fi}(\mathbb{A})$ for the set of finite sets of atoms. Write $\text{И} a.\ \Phi(a)$ for $\exists S \in \mathcal{P}_{fi}(\mathbb{A}).\ \forall a \notin S.\ \Phi(a)$. Then (5) above means

(6) $$\forall x \in X.\ \exists S \in \mathcal{P}_{fi}(\mathbb{A}).\ \forall a, b \notin S.\ (a\ b) \cdot x = x.$$

In fact there is a minimal **support** $S(x) \in \mathcal{P}_{fi}(\mathbb{A})$ such that

(7) $$a, b \notin S(x) \implies (a\ b) \cdot x = x.$$

Write $a \# x$ when $a \notin S(x)$.

This reflects the FreshML state: a 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>ty` is

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

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

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

$$f \mapsto \text{И}a.\, \langle a, fa\rangle$$
$$\langle a, x\rangle \mapsto \lambda b.(b\ a)x$$

(where $\sim$ takes equivalences over the choice of a in both maps). This duality between pairs and functions gives us (1) and (2).

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 *)
;
```

Ontology

I propose two ontological commitments in this slide. First, in `Proc` by use of `<Name>Proc` I propose the use of FM abstraction to model binding. In mathematical notation, $[a]P \in [\mathbb{A}]\Pi$ models $b.P$ in, say, $a[b]P$.

Second, in `Tran=<Name>(Act*Proc)` I propose to model π-calculus transitions by $\Pi \times [\mathbb{A}](Act \times \Pi)$. The slogan is:

> "Model freshly-generated names by binding."

I proposed this in [thempc] for the π-calculus. It has since been used also in the FreshML denotational semantics, see [frepbm], with great success.

Call a transition system $R \subseteq X \times Y$ **name-regular** when $\forall x R y.\, a\#x \Rightarrow a\#y$. The declaration of `Tran` makes it a name-regular transition system. Transitions

$$\begin{aligned} \overline{a}bP &\overset{\overline{a}b}{\to} \quad P \\ a[b]P &\overset{ab}{\to} \quad P \\ \nu[b]\overline{a}bP &\overset{\overline{a}b}{\to} \quad P \end{aligned} \tag{8}$$

are modelled by elements

```
( 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
```

The code which generates the transitions is. . .

```
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) $$\frac{P \overset{\nu m.\overline{a}b}{\longrightarrow} P'}{\nu[b]P \overset{\nu b.\overline{a}b}{\longrightarrow} P'}$$

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

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

I suggest FM abstraction and name-regular transition systems are an efficient and natural way of programming your process calculi.

I could examine `pi-ltsb-3` in more detail, but instead (have I got time left?); I indulge myself with some 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 $[\mathbb{A}\text{-}List]\alpha$.

```
(* 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 pattern 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) $$\frac{P_1 \overset{\nu c.\overline{a}b_1}{\longrightarrow} [l]Q_1 \quad P_2 \overset{\nu b_2.ab_2}{\longrightarrow} [l]Q_2}{P_1 \mid P_2 \overset{\nu c.\tau}{\longrightarrow} [c :: l](Q_1 \mid Q_2\{b_1/b_2\})}$$

Conclusions

Slogans are:

- Use the FM model of binding to specify syntax-up-to-binding.
- Use $\mathrm{FreshML}$ to program on it.
- Use FM binding to model generation of fresh names in transition systems.
- Use the abstraction monad to model restriction, in maths and programming.

I should write that up as a paper, shouldn't I? I have; in [thempc] and [thempc-3]. However FM techniques are better-understood and there is scope to re-state this case.