Restriction, Binding, and three presentations of the π-calculus

Murdoch J. Gabbay, December 2002

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

Sigh

Sigh. Yet another talk by Jamie on the π-calculus.

There's other things I can talk about but to be honest, this is what I want to tell you today. Aaargh! I can't help it!!

Sigh. Will he understand what he's talking about?

Not completely, but I'll try to make a good story of it.

Purpose of this talk

I will speak about two questions I have been trying to address

1. “What is the difference between binding and restriction?”
2. “What is it like to program in FreshML?” (Take a bow Mark)

in a series of FreshML programs called

```
pi-ltsb-1
pi-ltsb-2
pi-ltsb-3
pi-ltsb-4
```

Catchy, yes? Full lyrics available on my homepage. Let’s look at some code.

```
bindable_type Name             (* bound names *)
;
datatype Chan =                (* channel names *)
    Fn of string               (* free names *)
  | Bn of 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 Chan*Chan*Proc      (* out x y.(P) *)
  | In  of Chan*(<Name>Proc)  (* in x(y).(P)  *)
  | Tau of Proc                (* tau.(P) *)
  | Ina                        (* 0 *)
;
```

```
datatype Trn =           (* results of a transition step *)
    Actt of Proc
  | Acti of Chan*(<Name>Proc)
  | Acto of Chan*Chan*Proc
  | Actbo of Chan*(<Name>Proc);
```

$\mathcal{B} = \Pi + \mathbb{A} \times \delta\Pi + \mathbb{A}^2 \times \Pi + \mathbb{A} \times \delta\Pi$

```
val comm1_rule_helper : Trn*Trn -> (Trn option) =
   fn ( Acto(x1,y1,q1) , Acti(x2,<a2>q2) ) =>
      if x1=x2 then
         Some (Actt( Par(q1,rename(<a2>q2,y1)) ))
      else None
    | _ => None;
val close1_rule_helper : Trn*Trn -> (Trn option) =
   fn ( Actbo(x1,<a1>q1) , Acti(x2,<a2>q2) ) =>
      if x1=x2 then
         Some (Actt(Res(<a2>(Par(concrete (<a1>q1) at a2,
                                 q2)))))
      else None
    | _ => None;
```

Non-linear patterns would be nice, I'll come back to that later. E.g. compare:

```
val close1_rule_helper : Trn*Trn -> (Trn option) =
   fn ( Actbo(x1,<a1>q1) , Acti(x2,<a2>q2) ) =>
      if x1=x2 then
         Some (Actt(Res(<a2>(Par(concrete (<a1>q1) at a2,
                                 q2)))))
      else None
    | _ => None
;

val close1_rule_helper : Trn*Trn -> (Trn option) =
   fn ( Actbo(x,<a>q1) , Acti(x,<a>q2) ) =>
         Some (Actt(Res(<a>(Par(q1,q2)))))
    | _ => None
;
```

```
val comm1_rule : (Trn list) -> (Trn list) -> (Trn list) =
   mapMatrixPartial (fn trn1 => fn trn2 =>
                        comm1_rule_helper (trn1,trn2));

val close1_rule : (Trn list) -> (Trn list) -> (Trn list) =
   mapMatrixPartial (fn trn1 => fn trn2 =>
                        close1_rule_helper (trn1,trn2));

val rec trns_of : Proc -> (Trn list) =
 fn Ina => []
  | (Tau(p))      => [Actt p]
  | (Out(x,y,p))  => [Acto(x,y,p)]
    ...
  | (Par(p1,p2))  => (par1_rule p2 (trns_of p1))++
                     (comm1_rule (trns_of p1) (trns_of p2))++
                     (close1_rule (trns_of p1) (trns_of p2))++
                      ...
;
```

```
val open_rule_helper : <Name>Trn -> Trn option =
 fn  <n>(Acto(Fn s,Bn b,p'))   =>
    if n#b then None else
                Some (Actbo(Fn s,<n>p'))
  |  <n>(Acto(Bn c,Bn b,p'))   =>
    if n#b then None else
    if n#c then Some (Actbo(Bn c,<n>p')) else
                None
  |                          _ => None
;

val rec trns_of : Proc -> (Trn list) =
 fn Ina => []
  | (Tau(p))      => [Actt p]
  | (Out(x,y,p))  => [Acto(x,y,p)]
  | (In(x,p_hat)) => [Acti(x,p_hat)]
     ...
  | (Res(<n>p))   => open_rule (<n>(trns_of p))
     ...
end
;
```

```
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 *)
;
```

```
val comm_close_1_rule_helper :
 <Name>((Act*Proc)*(Act*Proc)) -> (Trn option) =
 fn <n>((Acto(x1,a1),q1),(Acti(x2,a2),q2)) =>
  if x1=x2 then
    if a1#n then
      Some (<n>(Actt,Par( q1,rename(<a2>q2,a1) )))
    else
      Some (<n>(Actt,Res(<n>(Par( q1,rename(<a2>q2,a1) ))) ))
 else None
| _ => None
;
val rec trns_of : Proc -> (Trn list) =
 fn Ina           => []
  | (Tau(p))      => [promoteAbs (Actt,p)]
  | (Out(x,y,p))  => [promoteAbs (Acto(x,y),p)]
  | (Par(p1,p2))  => let val trns1 = trns_of p1
                         and trns2 = trns_of p2
                     in  ...
                        (comm_close_1_rule trns1 trns2)++
                     end
```

```
val comm_close_1_rule : (Trn list) -> (Trn list) -> (Trn list) =
 mapMatrixPartial (
  fn trn1 => fn trn2 =>
   comm_close_1_rule_helper (pair_abs_abs_pair (trn1,trn2))
);

val pair_abs_to_abs_pair : (<Name>'x * <Name>'y) ->
                              <Name>('x * 'y) =
fn (x_hat,y_hat) => let fresh c:Name in
       (<c>(concrete x_hat at c, concrete y_hat at c)) end
;
```

```
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
;

val rec trns_of : Proc -> (Trn list) =
 fn Ina           => []
  | (Tau(p))      => [promoteAbs (Actt,p)]
  | (Out(x,y,p))  => [promoteAbs (Acto(x,y),p)]
  | (In(x,<n>p))  => [<n>(Acti(x,n),p)]
    ...
  | (Res(<n>p))   => open_rule n (trns_of p)
    ...
;
```

```
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
;
```

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

```
datatype ('@a,'b)am =
   amIn of 'b                 (* unit of the monad *)
 | amAb of <'@a>(('@a,'b)am); (* add an abstraction *)

(* Monad lifting function: abs >> f applies f to the abstracted value
in abs and adds abs's abstractions to the result. *)
infix >>;
val rec op>> : ('@a,'b)am * ('b -> ('@a,'c)am) -> ('@a,'c)am = fn
   (amIn x, f) => f x
 | (amAb(<a>y), f) => amAb(<a>(y >> f));

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

```
val comm_close_1_rule_helper :
   <Name>((Act*ProcNM)*(Act*ProcNM)) -> Trn option  =
   fn <n>( (Acto(x1,a1),q1am) , (Acti(x2,a2),q2am) ) =>
      if x1=x2 then Some (<a2>(Actt,amAb (<n>(
          (merge_am2 (q1am,q2am)) >> (fn (q1,q2) =>
              amIn (Par( q1,rename(<a2>q2,a1) ))
          )))))
      else None
    | _ => None;
```

. . . et comme il faudrait . . .

```
val comm_close_1_rule_helper :
   <Name>((Act*ProcNM)*(Act*ProcNM)) -> Trn option  =
   fn <n>( (Acto(x1,a1),<l>q1) , (Acti(x1,a2),<l>q2) ) =>
      Some <a2>(Actt , <n::l> Par(q1 , rename(<a2>q2,a1)) )
    | _ => None;
```

```
val rec trns_of : Proc -> (Trn list) =
 fn Ina           => []
  | (Tau(p))      => [promoteAbs (Actt,amIn p)]
  | (Out(x,y,p))  => [promoteAbs (Acto(x,y),amIn p)]
  | (In(x,<n>p))  => [<n>(Acti(x,n),amIn p)]
  | (Par(p1,p2))  => let val trns1 = trns_of p1
                             and trns2 = trns_of p2
                     in
                        (par1_rule p2 trns1)++
                        (par2_rule p1 trns2)++
                        (comm_close_1_rule trns1 trns2)++
                        (comm_close_2_rule trns1 trns2)
                     end
  | (Rep(pam))    =>
      listAM(pam,fn (l,p) => rep_rule (l,p,pam) (trns_of p))
;
```

We work with `Proc NM`. `trns_of` only ever gets applied as `pam >> (fn p => f(trns_of p))`.

```
val rep_rule_helper : Name list*Proc*ProcNM -> Trn -> Trn option =
 fn (l,p,pam) => fn
  <n>(Acto(a,b),qam) =>
   if list_in(a,l) then      None
   else if list_in(b,l) then Some (<n>(Acto(a,n),
    listAb(l,qam >> (fn q => amIn(Par(rename(<b>q,n),Rep(pam)))))))
   else                      Some (<n>(Acto(a,b),
    listAb(l,qam >> (fn q => amIn(Par(q,Rep(pam)))))))
| <n>(Acti(a,b),qam) =>
   if list_in(a,l) then None
   else Some (<n>(Acti(a,b),
    listAb(l,qam >> (fn q => amIn(Par(q,Rep(pam)))))))
| <n>(Actt,qam) => Some(<n>(Actt,
    listAb(l,qam >> (fn q => amIn(Par(q,Rep(pam)))))))
;

val rec listAb : '@a list * ('@a,'b) am -> ('@a,'b) am = fn
   ([],x) => x
 | (hd::tl,x) => amAb(<hd>(listAb(tl,x)))
;
```

. . . which is trying to be the following:

```
val rep_rule_helper : ProcNM -> Trn -> Trn option =
 fn <l>p => fn
  <n>(Acto(a,b),<l'>q) =>
   if list_in(a,l) then
      None
   else if list_in(b,l) then
      Some (<n>(Acto(a,n), <l@l'>Par(rename(<b>q,n),Rep(<l>p))
   else
      Some (<n>(Acto(a,b), <l@l'>Par(q,Rep(<l>p))
| <n>(Acti(a,b),<l'>q) =>
   if list_in(a,l) then
      None
   else
      Some (<n>( Acti(a,b) , <l@l'>Par(q,Rep(<l>p)) ))
| <n>(Actt,<l'>q) => Some(<n>( Actt , <l@l'>Par(q,Rep(<l>p)) ))
;
```

The original FM binding type-former is $[\mathbb{A}]X$. It has nice properties, for example:

(1) $$[\mathbb{A}]\mathbb{A} \cong \mathbb{A} + 1$$

(2) $$[\mathbb{A}]X \times [\mathbb{A}]Y \longrightarrow [\mathbb{A}](X \times Y)$$

(3) $$[\mathbb{A}](X \times Y) \longrightarrow [\mathbb{A}]X \times [\mathbb{A}]Y.$$

(Here's an obvious question: can we characterise the Schanuel Topos as a topos with an abstraction endofunctor satisfying nice properties such as those above. Matias Menni thought about that two years ago. Perhaps it's time to come back to the issue.)

Problem is, π-calculus restriction $\nu a.p$ is not an instance of this structure. For example $(\nu a.p \mid \nu a.q)$ is structurally congruent to $\nu a, b.(p \mid q\{a {\mapsto} b\})$ (for appropriate fresh b) and *not* to $\nu a.(p \mid q)$.

Restriction

So perhaps the original FM restriction type-former is $[\mathbb{A}\text{-}List]X$.
Scope extrusion is an instance of `'x NM >> (fn x => f(x))`, monadic application.

Tangential observation 1: Abstraction by lists with garbage collection of leading vacuous atoms commutes with finite limits and colimits but not infinite limits and colimits, and not with function spaces. This is my bet for a 'restriction' type-former.

(What is 'garbage collection'? $\nu ab.\, p \cong \nu a.\, p$ if $b \notin fn(p)$.)

Tangential observation 2: In FMG we could have abstraction by ω-*streams* of atoms. This has the properties both of a restriction *and* an abstraction. Perhaps that's why I thought it was so neat.

pi-ltsb-3

Obvious question: how well does $[\mathbb{A}\text{-}List]-$ model restriction? Can we axiomatise/program with abstraction and restriction type-formers using $[\mathbb{A}]-$ and (a relative of) $[\mathbb{A}\text{-}List]-$ as models?

Another question: can we apply programming like we saw in `pi-ltsb-4` to work by Cardelli *ed altri maestri* programming with tree structures with hiding.