Idioms for Interaction

— Invitation to Hacking in π-calculus —

Kohei Honda

Edinburgh University

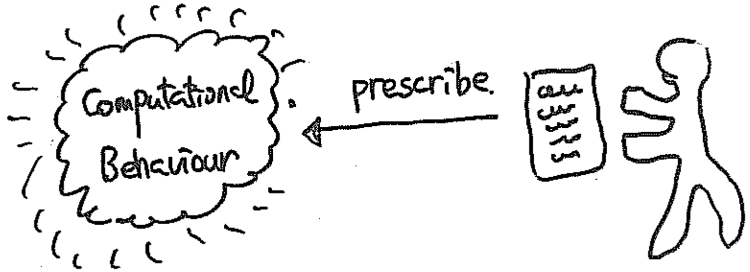
Content

1

Backgrounds.

On Programs and Programming (1).

- Programs: prescription of computational behaviours based on a certain abstraction.



- Programming Languages are tools which offer frameworks of abstraction for such activities — promoting or limiting them.
    - Imperative.
    - Functional
    - Logical
    - ⋮

On Programs and Programming (2).

- The most fundamental element of a PL in this context is a set of operations it is based on:

  Imperative: assignment, jump.

  Functional: β-reduction.

  Logical: unification.

- Another element is how we can combine, or structure, these operations:

  Imperative: sequential composition, if-then-else, while, procedures, module, ...

  Functional: application, product, union, recursion, modules, ...

P H value especially the latter.

```
      data  _stkl,48     stacks.
      data  _stkr,48
      data  _i,4         indices
      data  _j,4
      data  _l,4         left/right limits,
      data  _r,4
      data  _x,4         pivot
      data  _w,4         temporary value.
      data  _s,4         stack pointer
      data  _a,48        table to be sorted.

      mov $0,_s          Initialisation
      mov $0,_stkl
      mov $11,_stkr
                                        Top Loop.
L1:   mov _s,rax                        l:=stkl(s).
      mov _stkl(,rax,4),rcx
      mov rcx,_l
      mov _s,rax                        r:=stkr(s),
      mov _stkr(,rax,4),rcx
      mov rcx,_r
      dec _s
                                        Second Loop.
L2:   mov _l,rcx         i= l.
      mov rcx,_i
      mov _r,rcx         j:=r.
      mov rcx,_j
      mov _l,rdx         rdx:= l+r
      add _r,rdx
      mov rdx,rax        rdx:= (l+r)/2
      mov rax,rdx
      shr $31,rdx
      add rdx,rax
      mov rax,rdx
      sar $1,rdx
      mov _a(,rdx,4),rcx                x:= a(rdx).
      mov rcx,_x
                                        third loop
L3:   mov _i,rax                        rdx:=a(i).
      mov _a(,rax,4),rdx
      cmp rdx,_x
      jle L4                            If rdx≥x goto L4
      inc _i                            else i=i+1
      jmp L3                            goto L3. (loop).

L4:   mov _j,rax                        rdx=a(j).
      mov _a(,rax,4),rdx
      cmpl rdx,_x                       If rdx≤x goto L5
      jge L5                            else j:=j-1 and
      dec _j                            goto L4 (loop).
      jmp L4

L5:   mov _i,rax                        If i>j goto L6.
      cmpl rax,_j
      jl L6
      mov _i,rax                        w=a(i).
      mov _a(,rax,4),rcx
      mov rcx,_w
      mov _i,rax                        a(i)=a(j).
      mov _j,rdx
      mov _a(,rdx,4),rcx
      mov rcx,_a(,rax,4)
      mov _j,rax                        a(j)=w.
      mov _w,rcx
      mov rcx,_a(,rax,4)
      inc _i
      dec _j

L6:   mov _i,rax                        If i≤j loop at L3.
      cmp rax,_j                        else next.
      jl L7
      jmp L3

L7:   mov _i,rax                        if i≥r then L9
      cmp rax,_r
      jle L8
                                        else s:=s+1
      inc _s
      mov _s,rax                        stkl(s)=i;
      mov _i,rcx
      mov rcx,_stkl(,rax,4)
      mov _s,rax                        stkr(s)=r.
      mov _r,rcx
      mov rcx,_stkr(,rax,4)

L8:
      mov _l,rax                        if l≥r then
      cmp rax,_r                        next else
      jle L9                            to the second loop.
      jmp L2

L9:   cmp $0,_s                         if s=0 end
      jle L10                           else top loop.
      jmp L1

L10:  ret
```

```
Var  a: array[MAX] of int;

Procedure  sort(l, r : int);
  Var i, j, x: int;
   i:= l;  j:= r;
   x:= [(l+r) div2];
   repeat
     while  a[i] < x  do  i:= i+1  end
     while  a[j] > x  do  j:= j+1 end.
     if  i≤j  then  swap(i, j);  i:=i+1; j:=j-1 ; end
   until  i>j;
   if  l<j  then  sort(l, j);
   if  l<r  then  sort(i, r);
  end

Procedure  swap(i, j : int)
    Var  w: int;
         w:= a[i] ; a[i] := a[j] ; a[j]:=w;
end
```

• Choose a pivot.

Partition into two parts.

Recursively sort two parts.

$((\lambda xy.y(xxy))(\lambda xy.y(xxy)))\lambda q.\lambda l.$

$((\lambda x.x(\lambda xy.x))l)(\lambda x.x)$ } if l is nil then nil.

$(((\lambda xy.y(xxy))(\lambda xy.y(xxy))(\lambda c.\lambda xy.x((\lambda x.x(\lambda xy.x))x)y$

$((\lambda xy.\lambda z.z(\lambda xy.y)xy)((\lambda x.x(\lambda xyz.y))x)(c((\lambda x.x(\lambda xyz.z))x)y$ } concat.

$(q(\lambda xy.y(xxy))(\lambda xy.y(xxy))(\lambda f.\lambda px.((\lambda x.x(\lambda xy.x))x)(\lambda x.x))$

$(p((\lambda x.x(\lambda xyz.y))x))((\lambda xy.\lambda z.z(\lambda xy.y)xy)x$

$(f((\lambda x.x(\lambda xyz.z))x)))(f((\lambda x.x(\lambda xyz.z))x)))$ } SORT and Filter

$(\lambda y.(((\lambda xy.y(xxy))(\lambda xy.y(xxy)))\lambda f'.\lambda xy.((\lambda x.x\lambda xy.x)y)$ } λy. LTy. Car

$(\lambda xy.y)(((\lambda x.x\lambda xy.x)x)(\lambda xy.y)(f'((\lambda x.x\lambda xy.y)x)((\lambda x.x\lambda xy.y$

$y((\lambda x.x(\lambda xyz.y))l))((\lambda x.x(\lambda xyz.z))l))$ } cdr l

$((\lambda xy.\lambda z.z(\lambda xy.y)xy)((\lambda x.x(\lambda xyz.y))l)$ } Cons (Car l)

$(q((\lambda xy.y(xxy))(\lambda xy.y(xxy))(\lambda f.\lambda px.((\lambda x.x(\lambda xy.x))x)$

$(\lambda x.x)(p((\lambda x.x(\lambda xyz.y))x))((\lambda xy.\lambda z.z(\lambda xy.y)xy)x$

$(f((\lambda x.x(\lambda xyz.z))x)))(f((\lambda x.x(\lambda xyz.z))x)))$ } SORT and Filter

$(\lambda y.((\lambda xy.y(xxy))(\lambda xy.y(xxy)))\lambda f''.\lambda xy.((\lambda x.x\lambda xy.x)x)$

$(((\lambda x.x\lambda xy.x)y)(\lambda xy.x)(\lambda xy.y))$

$(((\lambda x.x\lambda xy.x)y)(\lambda xy.x)(f''((\lambda x.x\lambda xy.y)x)$ } λy. M≥y Car l

$((\lambda x.x\lambda xy.y)y)))y((\lambda x.x(\lambda xyz.y))l))((\lambda x.x(\lambda xyz.z))l))))).$ } cdr l

quicksort in pure lambda.

$$
\begin{aligned}
&Y\,(\lambda f.\ \lambda l. \\
&\quad (\mathrm{Isnil}\ l) \\
&\quad (\mathrm{Concat}\ (f\ \mathrm{Filter}\ (\lambda y.\ \mathrm{LT}\,y\,(\mathrm{Car}\,l)) \\
&\qquad\qquad (\mathrm{Cdr}\,l)) \\
&\qquad (\mathrm{Cons}\ (\mathrm{Car}\,l) \\
&\qquad\quad (f\ \mathrm{Filter}\ (\lambda y.\ \mathrm{Me}\,y \\
&\qquad\qquad (\mathrm{Car}\,l) \\
&\qquad\qquad (\mathrm{Cdr}\,l\,)))))
\end{aligned}
$$

$I = \lambda x.x \quad T = \lambda xy.x \quad F = \lambda xy.y \quad Y = (\lambda xy.y(xxy))(\lambda xy.y(xxy))$

$\mathrm{Cons} = \lambda xy.\lambda z.zFxy \qquad \mathrm{Isnil} = \lambda x.xT$

$\mathrm{Car} = \lambda x.x(\lambda xyz.y) \qquad \mathrm{Cdr} = \lambda x.x(\lambda xyz.z)$

$\mathrm{Concat} = Y\,(\lambda c.\lambda xy.\ x\ (\mathrm{Isnil}\ x)\,y\ (\mathrm{Cons}\ (\mathrm{Car}\,x)(c(\mathrm{Cdr}\,x)y)$

$\mathrm{Filter} = Y(\lambda f.\lambda px.(\mathrm{Isnil}\ x)\,I(p(\mathrm{Car}\,x))(\mathrm{Cons}\ x\ (\mathrm{Filter}\,(\mathrm{Cdr}\,x\,)))$
$\qquad\qquad (\mathrm{Filter}\,(\mathrm{Cdr}\,x))$

$\mathrm{Iszero} = \lambda x.\,xT \qquad \mathrm{Pred} = \lambda x.\,xF$

$\mathrm{LT} = Y\,(\lambda f.\ \lambda xy.\ (\mathrm{Iszero}\ y)\ F\,((\mathrm{Iszero}\ x)\ T\,(f(\mathrm{Pred}\ x)(\mathrm{Pred}\ y)))$

$\mathrm{ME} = Y\,(\lambda f.\ \lambda xy.\ (\mathrm{Iszero}\ x)((\mathrm{Iszero}\ y)\ I\ F)((\mathrm{Iszero}\ y)(T)$
$\qquad (f\,(\mathrm{Pred}\ x)(\mathrm{Pred}\ y)))$

```
fun qs nil:int list = nil
  | qs (x::r) = let val small =
                        filter (fn y => y<x) r
                    and large =
                        filter (fn y => y>=x) r
                in qs small @ [x] @ qs large
                end

fun filter p nil = nil
  | filter p (x::r) =
        if p x then x:: filter p r
               else filter p r
```

2

Preparing for Joyful Hacking in π-calculus.

Syntax

Let a, b, c... be <u>names</u>. We shall work with the following syntax.

$P ::=$	$\overline{a}b$	Message.
$\mid$	$a\underline{x}.P$	receptor. $a(x).P$
$\mid$	$P \mid Q$	parallel composition.
$\mid$	$\underline{a} \blacktriangleright P$	name hiding. $(\nu a)P$
$\mid$	$\text{letrec } X(x) = P \text{ in } Q.$	recursion.
$\mid$	$\emptyset.$	inaction.

Comments on Syntax.

(1) Message $\overline{a}b$ comes from $\overline{a}b.\emptyset$.

(2) $ax.P$ is an "a-pointed (named, located) name abstraction over P." cf. $\lambda x.M$.

(3) Recursion and replication $!P$ are interdefinable. We use:

$$!ax.P \stackrel{\text{def}}{=} \text{letrec } X(a\vec{v}) = ax.(P \mid X(a\vec{v})) \text{ in } X(a\vec{v}).$$

Structural Rules.

* $\equiv$ is the smallest congruence closed under:

$$P \equiv Q \quad \text{whenever } P \equiv_\alpha Q.$$

$$P \mid \emptyset \equiv P \quad P \mid Q \equiv Q \mid P \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$a \blacktriangleright \emptyset \equiv \emptyset \quad aa \blacktriangleright P \equiv P \quad ab \blacktriangleright P \equiv ba \blacktriangleright P$$

$$a \blacktriangleright P \mid Q \equiv a \blacktriangleright (P \mid Q) \quad a \notin FV(Q).$$

$$(\text{letrec } X(\vec{x}) = P \text{ in } Q) \mid R \equiv \text{letrec } X(\vec{x}) = P \text{ in } (Q \mid R)$$
$$X \notin FV(R).$$

$$\text{letrec } X(\vec{x}) = P \text{ in } Q \equiv Q[X(\vec{b}) \mapsto P[\vec{b}/\vec{x}]]$$

* Specifically $!ax.P \equiv ax.(P \mid !ax.P)$ $(a \neq x)$.

Reduction

* Define one step reduction $\to$ by:

(COM) $\quad ax.P \mid \bar{a}b \to P[b/x]$

(PAR) $\quad P \to P'$ then $P \mid Q \to P' \mid Q$.

(RES) $\quad P \to P'$ then $a \triangleright P \to a \triangleright P'$.

(STR) $\quad P \equiv P'$ $P' \to Q'$ $Q' \equiv Q$ then $P \to Q$.

* Then $\twoheadrightarrow \stackrel{def}{=} \to^{*} \cup \equiv$.

☆ Note the similarity / differences with $(\lambda x.M)N \to M[N/x]$.
name passing / term passing.
shared / non-shared.

Examples of Reduction.

(1) $\underline{a}x.\bar{b}x \mid \underline{\bar{a}}v \to \bar{b}v$.

(2) $\underline{a}x.\bar{b}x \mid \bar{c}w \mid \underline{\bar{a}}v \equiv ax.\bar{b}x \mid \bar{a}v \mid \bar{c}w$
$\to \bar{b}v \mid \bar{c}w$.

(3) $(b \triangleright \underline{a}x.\bar{x}b) \mid \underline{\bar{a}}b \equiv c \triangleright (ax.\bar{x}c \mid \bar{a}b)$
$\to c \triangleright \bar{b}c$.

(4) $\underline{a}x.\bar{b}x \mid \underline{\bar{a}}v \mid \underline{\bar{a}}w \to \bar{b}v \mid \bar{a}w$
$\searrow \bar{b}w \mid \bar{a}v$

(5) $\underline{a}x.(\bar{b}x \mid \bar{c}x) \mid \underline{\bar{a}}v \to \bar{b}v \mid \bar{c}v$

(6) $\underline{a}x.0 \mid \underline{\bar{a}}v \to 0$

Equations (1)

* Using the labelled transition relation induced by such rules as:

(IN) $ax.P \xrightarrow{\downarrow av} P[v/x]$

(OUT) $\bar{a}v \xrightarrow{\uparrow av} \emptyset$.

We have the usual weak bisimilarity $\approx$. Then $\approx$ is closed under name substitution and) a congruence.

$$\begin{array}{ccc} P & \xrightarrow{l} & P' \\ \approx & & \approx \\ Q \twoheadrightarrow & \xrightarrow{\hat{l}} & \twoheadrightarrow Q' \end{array}$$

Equations (2)

* More "syntactic" equalities:

<u>Definition.</u>

P is a^θ-pointed if $P \not\rightarrow$ and a^θ is the only active occurrence in P.

e.g. $b\triangleright(\underset{\sim}{\bar{a}}b \mid by.Q)$ is a^--pointed.

$ax.P$

$\underset{\sim}{a}x.P$ and $c\triangleright(\underset{\sim}{a}x.\bar{b}c \mid cy.Q)$ are a^+-pointed

$\underset{\sim}{\bar{a}}b \mid \bar{c}e$ or $c\triangleright(cx.\emptyset \mid \bar{c}b)$ are NOT pointed.

Equation (3).

Definition

Relations $\xrightarrow{\beta}$ and $\gg_{GC}$ are defined by:

$$\frac{P\langle a^+\rangle \mid P'\langle a^-\rangle \to R}{a\nu(P\langle a^+\rangle \mid P'\langle a^-\rangle) \xrightarrow{\beta} a\nu R}$$

$\approx$

and close under $\equiv$ and context.

$$a\nu P\langle a^b\rangle \gg_{GC} \emptyset$$

$\approx$

$$a\nu \bar{a}b = \emptyset$$

Prop.

$\xrightarrow{\beta} \subseteq \approx$ and $\gg_{GC} \subseteq \approx$.

Definition.

$\overset{\beta}{\Rightarrow} \overset{\text{def}}{=} \xrightarrow{\beta}^* \cup \equiv$, $\ggg_{GC} \overset{\text{def}}{=} \gg_{GC}^* \cup \equiv$

3

Primitives for Interaction.

(Basic Techniques in Mobile Hacking.)

Sequentialisation (1)

- As a first step, we wish to realise sequencing in communication:

$$a:(x_1 \dots x_n).P \mid \overline{a}:[v_1 \dots v_n] \to P[v_1 \dots v_n / x_1 \dots x_n]. \quad (*)$$

cf. currying in λ-calculus.

$$\lambda(x_1 \dots x_n).M \equiv \lambda x_1 \lambda x_2 \dots \lambda x_n.M.$$

- How can we make $(*)$ from:

$$ax.P \mid \overline{a}v \to P[v/x] \ ?$$

Sequentialisation (2)

- Answer ([MPW89, HT91]);

<u>Definition</u> Let $c, c', y, z, \dots$ be fresh below.

$$a:(x_1 \dots x_n).P \stackrel{\text{def}}{=} c \triangleright az(\overline{z}c \mid cx_1.(\overline{z}c \mid cx_2.(zc \mid cx_3.\dots P)\dots)$$

$$\overline{a}:[v_1 \dots v_n].P \stackrel{\text{def}}{=} c' \triangleright (\overline{a}c \mid c'y.(\overline{y}v_1 \mid c'y.(\overline{y}v_2 \mid c'y.\overline{y}v_3 \mid \dots P)\dots))$$

<u>Proposition</u>

$a:(x_1 \dots x_n).P$ and $\overline{a}:[v_1 \dots v_n].P$ are a^{-}/a^{+}-pointed.

Moreover;

$$a:(x_1 \dots x_n).P \mid \overline{a}:[v_1 \dots v_n].Q$$

$$\to \stackrel{\beta}{\twoheadrightarrow} P[v_1 \dots v_n / x_1 \dots x_n] \mid Q.$$

How Sequentialisation Works.

$$a:(x_1 x_2).P \mid \bar{a}:[v_1 v_2].Q$$

$$\stackrel{\text{def}}{=} c \triangleright az.(\bar{z}c \mid cx_1.(\bar{z}c \mid cx_2.P)) \mid$$
$$c' \triangleright (\bar{a}c' \mid c'y.(\bar{y}v_1 \mid c'y.(\bar{y}v_2\ Q)))$$

$$\equiv cc' \triangleright \begin{pmatrix} \underset{\sim}{az}.(\bar{z}c \mid cx_1.(\bar{z}c \mid cx_2.P)) \mid \\ \underset{\sim}{\bar{a}c'} \mid c'y.(\bar{y}v_1 \mid c'y.(\bar{y}v_2 \mid Q)) \end{pmatrix}$$

• by c, c' fresh.

$$\rightarrow cc' \triangleright \begin{pmatrix} \underset{\sim}{\bar{c}'c} \mid cx_1.(\bar{c}'c \mid cx_2.P) \mid \\ \underset{\sim}{c'y}.(\bar{y}v_1 \mid c'y.(\bar{y}v_2 \mid Q)) \end{pmatrix}$$

• $c \triangleright (\bar{c}'c, cx_1 \ldots)$ and $c'y.(\bar{y}v_1, \ldots)$ are both pointed.

$$\xrightarrow{\beta} cc' \triangleright \begin{pmatrix} \underset{\sim}{cx_1}.(\bar{c}'c \mid cx_2.P) \mid \\ \underset{\sim}{\bar{c}v_1} \mid (c'y.(\bar{y}v_2 \mid Q)) \end{pmatrix}$$

• similarly.

$$\xrightarrow{\beta} cc' \triangleright \begin{pmatrix} \underset{\sim}{\bar{c}'c} \mid cx_2.P[v_1/x_1] \mid \\ \underset{\sim}{c'y}.(\bar{y}v_2 \mid Q) \end{pmatrix} \xrightarrow{\beta}\xrightarrow{\beta}\equiv \begin{pmatrix} P[v_1 v_2/x_1 x_2] \mid \\ Q. \end{pmatrix}$$

Branching (1).

• Next problem: Can we realise :

$$a:[\text{left}: P] \,\&\, [\text{right}: Q] \mid \bar{a}:\text{left}$$
$$\rightarrow P,$$

$$a:[\text{left}: P] \,\&\, [\text{right}: Q] \mid \bar{a}:\text{right}$$
$$\rightarrow Q.$$

I.e. branching/selection or method invocation ?

cf. Case constructs in imperative programming, sums in λ-calculi.

Branching (2).

- Answer ([Milner 90, HT 91]):

<u>Definition</u> Let c, c_1, c_2, z, y_1, y_2 be fresh below.

$$\overset{\overline{a}}{a}:[(x).P_1] \,\&\, [(y).P_2] \overset{\mathrm{def}}{=} c_1c_2\nu\ az.\ \overset{\overline{a}:[c_1c_2]}{\overline{z}:[c_1c_2]}.\begin{pmatrix} c_1(x).P_1 \\ c_2(y).P_2 \end{pmatrix}$$

$$\begin{cases} \overset{a}{\overline{a}}:in_1([v].P) \overset{\mathrm{def}}{=} c\nu\left(\overline{ac} \,\middle|\, \overset{a:(y_1y_2)}{c}:(y_1y_2).\ \overline{y_1}:[v].P\right) \\ \overset{a}{\overline{a}}:in_2([v].P) \overset{\mathrm{def}}{=} c\nu\left(\overline{ac} \,\middle|\, \overset{a:(y_1y_2)}{c}:(y_1y_2).\ \overline{y_2}:[v].P\right) \end{cases}$$

red: variant.

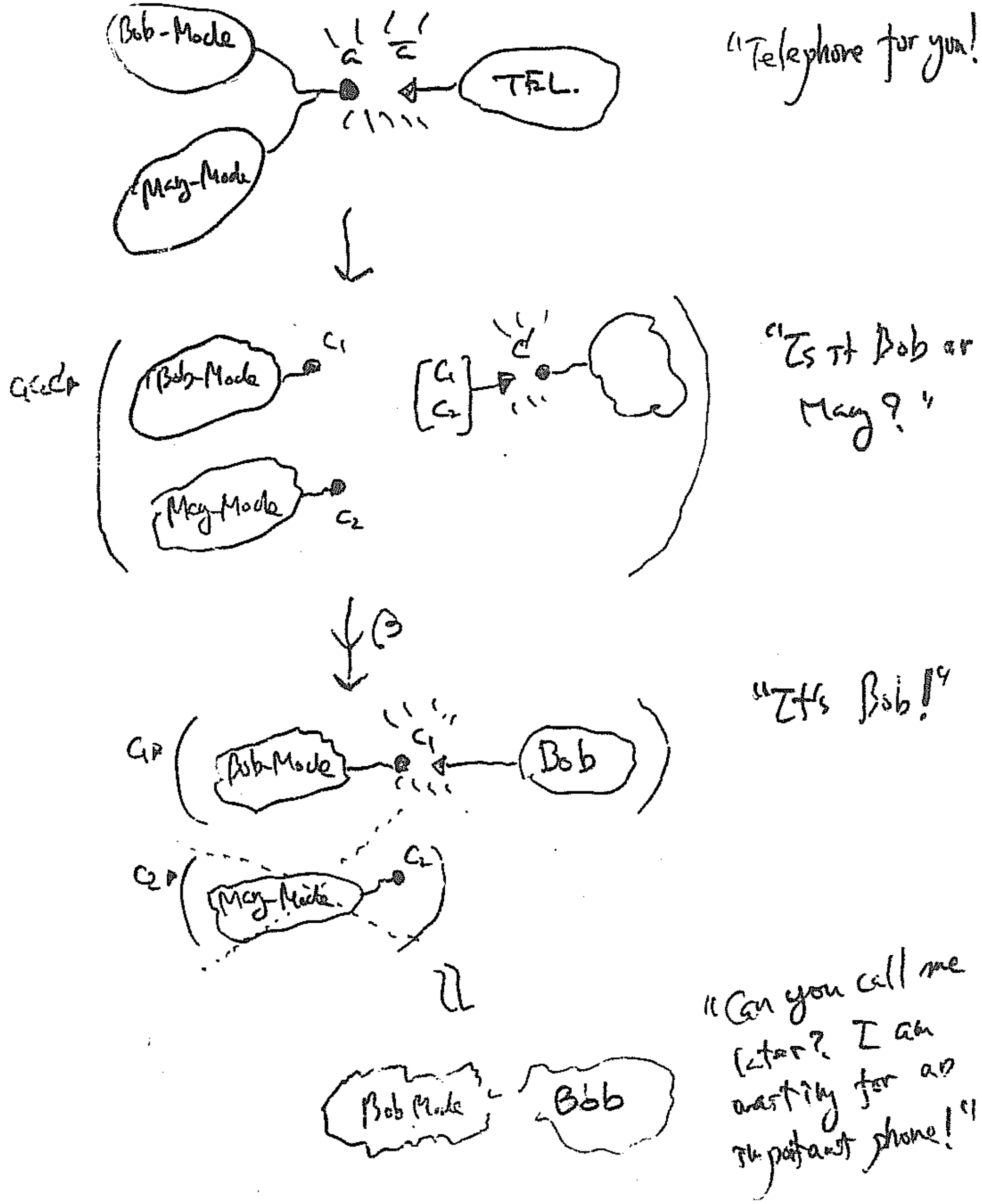
<u>Proposition</u>

$a:[(x).P_1]\&[(y).P_2]$ is a-pointed and $\overline{a}:in_i([v]P)$ is $\overline{a}$-pointed. Moreover:

$$a:[(x).P_1]\&[(y).P_2] \mid \overline{a}:in_i([v].Q)$$

$$\rightarrow \overset{\beta}{\rightarrow} \gg_{ac} P_i[v/x] \mid Q.$$

How Branching Works.



How Branching Works

$$T(a) = a(c_1 c_2).\, \overline{c_1} \qquad = a : \mathrm{inl}(.0)$$
$$F(a) = a(c_1 c_2).\, \overline{c_2} \qquad = a : \mathrm{inr}(.0)$$

$\mathrm{IfThenElse}(a, P_1, P_2)$

$$= (c_1 c_2)\, \overline{a}[c_1 c_2].(c_1.P_1 \mid c_2.P_2)$$
$$= \overline{a} : [P_1] \& [P_2]$$

$$T(a) \mid \mathrm{Ifthenelse}(a, P_1, P_2)$$

$$\equiv a(c_1 c_2).\overline{c_1} \mid (c_1 c_2)\, \overline{a}[c_1 c_2](c_1.P_1 \mid c_2.P_2)$$

$$\to (c_1 c_2)(\overline{c_1} \mid c_1.P_1 \mid c_2.P_2)$$

$$\xrightarrow{\beta} P_1 \mid (c_2)\, c_2.P_2$$

$$\Rightarrow_{GC} P_1.$$

Variants of Branching.

* The following variant is also useful.

$$\overline{a} : [(x_1).P_1] \& [(x_2).P_2] \overset{\mathrm{def}}{=} (c_1 c_2)\left(\overline{a} : [c_1 c_2] \;\middle|\; \begin{matrix} c_1 : (\tilde{x_1}).P_1 \\ c_2 : (\tilde{x_2}).P_2 \end{matrix}\right)$$

$$\begin{cases} a : \mathrm{inl}([\tilde{v}].P) \overset{\mathrm{def}}{=} a:(y_1 y_2).\, \overline{y_1}[\tilde{v}].P \\ a : \mathrm{inr}([\tilde{v}].P) \overset{\mathrm{def}}{=} a:(y_1 y_2).\, \overline{y_2}[\tilde{v}].P \end{cases}$$

* We also use labels:

$$a : [\underline{l_1} : (x).P_1,\ \underline{l_2} : (y).P_2]$$
$$\overline{a} : \underline{l_1}[\tilde{v}].P$$

for intuitive understanding.

Encoding Elementary Arithmetic. (1)

* How ARITHMETICAL OPERATIONS look like in the interaction paradigm?

* We start from natural numbers.

$$\overline{0}(a) \overset{df}{=} \,!a : \mathrm{inl}[.\emptyset]$$

$$\mathrm{succ}(ap) \overset{df}{=} \,!a : \mathrm{inr}[(p).\emptyset]$$

$$\overline{1}(a) \overset{df}{=} (p)(\mathrm{succ}(ap) \mid \overline{0}(p))$$

$$\overline{2}(a) \overset{df}{=} (p)(\mathrm{succ}(ap) \mid \overline{1}(p))$$

$$\equiv (pp')(\mathrm{succ}(ap) \mid \mathrm{succ}(pp') \mid \overline{0}(p'))$$

$$\vdots$$

$$\overline{N'}(a) \overset{df}{=} (p)(\mathrm{succ}(ap) \mid \overline{N}(p))$$

* A natural number, when invoked, tells whether it is zero or not, and if not who is its predecessor.

Encoding Elementary Arithmetics (2)

* We can duplicate a natural number by "decoding" its structure.

$$\mathrm{dupl}(ab) \overset{df}{=} \overline{a} : [.\,\overline{0}(b)] \,\&\, [(x).\,(p)(\mathrm{succ}(bp) \mid \mathrm{dupl}(xp))]$$

Also let:

$$\overline{\mathrm{pred}}(ab) \overset{df}{=} \overline{a} : [.\,\overline{0}(b)] \,\&\, [(x).\,\mathrm{dupl}(xb)]$$

$$\underline{\text{if}}\ a\ \underline{\text{then}}\ P_1\ \underline{\text{else}}\ P_2$$

$$\overset{df}{=} \overline{a} : [.\,P_1] \,\&\, [(z).\,P_2] \quad (z \text{ fresh})$$

Prop

- $\mathrm{dupl}(ab) \mid \overline{N}(a) \rightarrow \overset{c}{\twoheadrightarrow} \gg \overline{N}(a) \mid \overline{N}(b)$
- $\overline{\mathrm{pred}}(ab) \mid \overline{N}(a) \rightarrow \overset{c}{\twoheadrightarrow} \gg \overline{N}(a) \mid \overline{N-1}(b)$
- $\begin{cases} (\text{if } a \text{ then } P_1 \text{ else } P_2) \mid \overline{0}(a) \rightarrow \overset{c}{\twoheadrightarrow} \gg P_1 \mid \overline{0}(a) \\ (\text{if } a \text{ then } P_1 \text{ else } P_2) \mid \overline{N'}(a) \rightarrow \overset{c}{\twoheadrightarrow} \gg P_2 \mid \overline{N'}(a). \end{cases}$

Encoding Elementary Arithmetics. (3).

* Addition and Multiplication.

$$\overline{add}(a_1 a_2 b) \stackrel{\text{def}}{=} \underline{\text{if}}\ a_1\ \underline{\text{then}}\ \text{dupl}(a_2 b)$$

$$\underline{\text{else}}\ pr\triangleright\left(\overline{pred}(a_1 p) \mid \overline{add}(p a_2 r) \mid \overline{succ}(b r)\right)$$

$$\overline{mult}(a_1 a_2 b) \stackrel{\text{def}}{=} \underline{\text{if}}\ a_1\ \underline{\text{then}}\ \overline{0}(b)$$

$$\underline{\text{else}}\ pr\triangleright\left(\overline{pred}(a_1 p) \mid \overline{mult}(p a_2 r) \mid \overline{and}(r a_2 b)\right)$$

* Indeed, e.g.

$$a_1 a_2\triangleright\left(\overline{mult}(a_1 a_2 b) \mid \overline{N}(a_1) \mid \overline{M}(a_2)\right)$$

$$\overset{\beta}{\twoheadrightarrow} \ggg_{ac}\ \overline{N\times M}(b).$$

In this way any computable function is representable.

Some More Expressions.

* Predicate over natural numbers:

$$\overline{eq}(a_1 a_2 b) \stackrel{\text{def}}{=} \text{if}\ a_1\ \text{then}$$

$$(\text{if}\ a_2\ \text{then}\ \overline{T}(b)\ \text{else}\ \overline{F}(b))$$

else

$$(\text{if}\ a_2\ \text{then}\ \overline{F}(b)$$

$$\text{else}\ p_1 p_2\triangleright\left(\overline{pred}(a_1 p_1) \mid \overline{pred}(a_2 p_2) \mid \overline{eq}(p_1 p_2 b)\right))$$

$$\overline{le}(a_1 a_2 b) \stackrel{\text{def}}{=} \text{if}\ a_1\ \text{then}\ \overline{T}$$

$$\text{else if}\ a_2\ \text{then}\ \overline{F}(b)$$

$$\text{else}\ p_1 p_2\triangleright\left(\overline{pred}(a_1 p_1) \mid \overline{pred}(a_2 p_2) \mid \overline{eq}(p_1 p_2 b)\right).$$

with $\overline{T}(a) \stackrel{\text{def}}{=} \overline{0}(a)$, $\overline{F}(a) \stackrel{\text{def}}{=} \overline{1}(a)$.

Passing Values, Passing Names. (1)

* Regard $\overline{3}(b)$ etc. as constant agents.

Also note that it is stateless and persistent.

$$\overline{3}(a) \mid \overline{pred}(ab) \to \approx \overline{3}(a) \mid \overline{2}(b)$$

* Now write, for such agents,

$$\overline{a}G \overset{\text{def}}{=} c\nu\left(\overline{a}c \mid G(c)\right)$$

e.g. $\overline{a}3 \overset{\text{def}}{=} c\nu(\overline{a}c \mid \overline{3}(c))$ etc. Then for R with "good behaviour",

$$\overline{a}\nu\left(c\nu\left(\overline{a_1}\underset{\sim}{c} \mid \overline{a_2}\underset{\sim}{c} \mid \cdots \mid \overline{a_n}\underset{\sim}{c} \mid \underset{\sim}{G(c)}\right) \mid R\right)$$

$$\approx \overline{a}\nu\left(\overline{a_1}G \mid \overline{a_2}G \mid \ldots \overline{a_n}G \mid R\right).$$

Passing Values, Passing Names. (2).

* This allows us to write (under a mild consistency condition):

$$\overline{a}:[3,5].P$$

$$\overline{a}:[2+9].P$$

$$a:(X_1X_2).\ \overline{b}:[X_1+X_2]$$

$$\vdots$$

etc.

A Simple NAT-Cell.

* A cell is a simple stateful agent with "read"/"write" options.

$$\mathrm{Cell}(aN) \stackrel{\mathrm{def}}{=} a : [\underline{?\mathrm{read}} : [N].\, \mathrm{Cell}(aN)$$
$$\underline{?\mathrm{write}} : (X).\, \mathrm{Cell}(aX)]$$

$$\mathrm{Reader}(aXP) \stackrel{\mathrm{def}}{=} \overline{a} : \underline{!\mathrm{read}}((X).\, P)$$
$$\mathrm{Writer}(aM) \stackrel{\mathrm{def}}{=} \overline{a} : \underline{!\mathrm{write}}([M])$$

Then

$$\mathrm{Cell}(aN) \mid \mathrm{Read}(aXP)$$
$$\rightarrow \xrightarrow{\beta} \gg \mathrm{Cell}(aN) \mid P[N/X]$$

$$\mathrm{Cell}(a\underline{N}) \mid \mathrm{Writer}(aM)$$
$$\rightarrow \xrightarrow{\beta} \gg \mathrm{Cell}(a\underline{M})$$

Abstracting Interaction [Milner 91, Abramsky 90]

* Abstracting sequencing:

$$a : (x_1 .. x_n).\, P \Rrightarrow a^{+} : \underset{\sim}{\downarrow}(\alpha_1 ... \alpha_n)$$
$$\overline{a} : [v_1 .. v_n].\, P \Rrightarrow a^{-} : \underset{\sim}{\uparrow}(\alpha_1 .. \alpha_n)$$

with:

$$\overline{\downarrow(\alpha_1 .. \alpha_n)} = \uparrow(\alpha_1 .. \alpha_n) \qquad \overline{\uparrow(\alpha_1 .. \alpha_n)} = \downarrow(\alpha_1 .. \alpha_n).$$

* Abstracting branching/selection.

$$a : [(x).\, P_1] \,\&\, [(y).\, P_2] \Rrightarrow a^{+} : \downarrow(\alpha_1 .. \alpha_n) \,\underset{\sim}{\&}\, \downarrow(\beta_1 .. \beta_n)$$

$$\left\{ \begin{array}{l} \overline{a} : \mathrm{inl}([v].\, P) \\ \overline{a} : \mathrm{inr}([w].\, Q) \end{array} \right. \Rrightarrow a^{-} : \uparrow(\alpha_1 .. \alpha_n) \underset{\sim}{\oplus} \uparrow(\beta_1 .. \beta_n)$$

with

$$\overline{\alpha_1 \,\&\, \alpha_2} = \overline{\alpha_1} \oplus \overline{\alpha_2}$$

$$\overline{\alpha_1 \oplus \alpha_2} = \overline{\alpha_1} \,\&\, \overline{\alpha_2}.$$

Types for Natural Numbers.

* Because

$$\overline{0}(a) \stackrel{\text{df}}{=} !a : \mathrm{inl}(.\emptyset)$$

a natural number should have a type:

$$a^{+} : \mathbb{1} \oplus \ldots\ldots$$

* Because

$$\overline{\mathrm{succ}}(ab) \stackrel{\text{df}}{=} !a : \mathrm{inr}([b].\emptyset)$$

we now get

$$a^{+} : \mathbb{1} \oplus \uparrow\alpha$$

but α is also a natural number, so that we get:

$$a^{+} : \mathrm{NAT} \stackrel{\text{df}}{=} \mu\alpha.\, \mathbb{1} \oplus \uparrow\alpha. = \mathbb{1} \oplus \uparrow\mathrm{NAT}.$$

* Decoders etc. then have a type:

$$a^{-} : \overline{\mathrm{NAT}} = \mu\alpha.\, \mathbb{1} \& \downarrow\alpha. = \mathbb{1} \& \downarrow\mathrm{NAT}$$

$$\overline{a} : [.\ \ldots\ ] \& [(x).\ \ldots\ ]$$

Intermediate Summary: Two Idioms.

(1) Sequential Name Passing.

$$a!(x_1 \ldots x_n).P \mid \overline{a} : [v_1 \ldots v_n].Q$$

$$\rightarrow\approx P[\tilde{v}/\tilde{x}] \mid Q$$

(2) Branching / Selection.

$$a : [(\tilde{x}).P_1] \& [(\tilde{y}).P_2] \mid \overline{a} : \mathrm{inl}([\tilde{v}].Q)$$

$$\rightarrow\approx P_1[\tilde{v}/\tilde{x}] \mid Q.$$

cf. $(\lambda(x_1..x_n).M)\langle N_1,..,N_n\rangle \twoheadrightarrow M[\tilde{N}/\tilde{x}]$

$$\begin{pmatrix}\text{case } x \text{ of} \\ \mathrm{inl}(y) \Rightarrow M_1 \\ \mathrm{inr}(z) \Rightarrow M_2\end{pmatrix} \mathrm{inl}(N) \twoheadrightarrow M_1[N/y]$$

- dyadicity.
- sharing / interference.

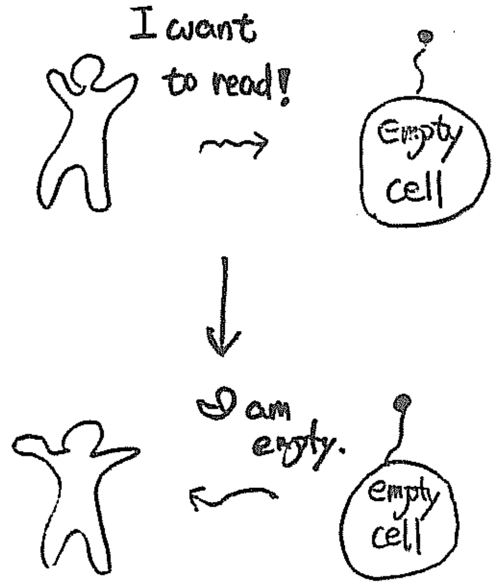
4

Combining Interaction.

(Further Techniques on Mobile Hacky).

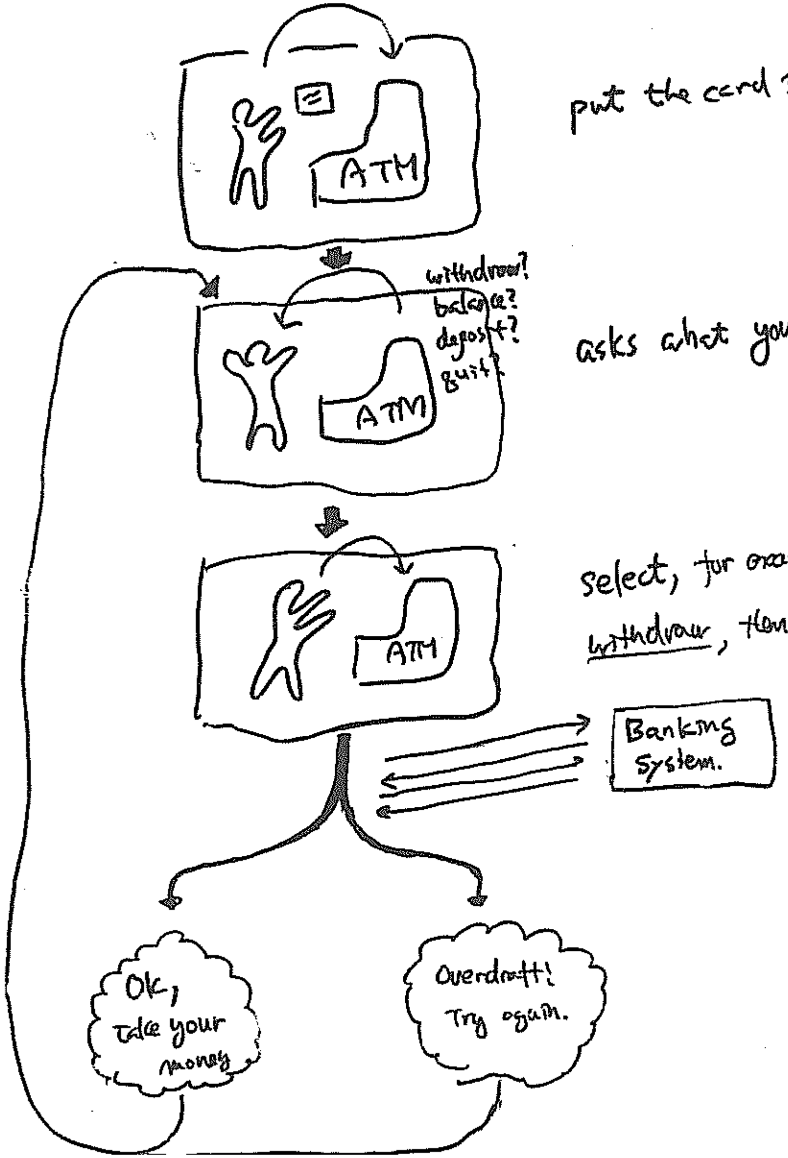
Combining Actions. (1)

- Realising a sequence of high-level actions.



Combining Actions (2)

- More complex series of actions.



put the card in.

asks what you want.

select, for example, withdraw, then key-in the amount.

Constructs for Combining Interactions. (1)

(1) Initiating Interaction. ("establish a channel").

$$a\langle z\rangle :: Act_1 \mid \bar{a}\langle z\rangle :: Act_2$$

$$\rightarrow \xrightarrow{0} z \triangleright (Act_1 \mid Act_2)$$

* z is used as the context of a "session".

(2) Passing Values.

$$z \triangleright (z?(x); Act_1 \mid z![v]; Act_2)$$

$$\xrightarrow{0} z \triangleright (Act_1[v/x] \mid Act_2)$$

(3) Branching / Selection

$$z \triangleright (z?[Act_1] \& [Act_2] \mid z! in_i; Act)$$

$$\xrightarrow{0} z \triangleright (Act_i \mid Act).$$

Constructs for Combining Interactions (2).

* Of course all can be realised by the basic calculus.

$$[\![a(z) :: \mathrm{Act}]\!] \stackrel{\mathrm{def}}{=} a:(z).\ [\![\mathrm{Act}]\!]$$

$$[\![\overline{a}(z) :: \mathrm{Act}]\!] \stackrel{\mathrm{def}}{=} z \blacktriangleright \overline{a}:[z].\ [\![\mathrm{Act}]\!].$$

$$[\![z?(x); \mathrm{Act}]\!] \stackrel{\mathrm{def}}{=} z:(x).[\![\mathrm{Act}]\!]$$

$$[\![z!(v); \mathrm{Act}]\!] \stackrel{\mathrm{def}}{=} \overline{z}:[v].\ [\![\mathrm{Act}]\!]$$

$$[\![z?[\mathrm{Act}_1 \,\&\, \mathrm{Act}_2]]\!] \stackrel{\mathrm{def}}{=} c_1 c_2 \blacktriangleright \overline{z}:[c_1 c_2].\begin{pmatrix} c_1 : [\![\mathrm{Act}_1]\!] \\ c_2 : [\![\mathrm{Act}_2]\!] \end{pmatrix}$$

$$[\![z!\mathrm{in}_i; \mathrm{Act}]\!] \stackrel{\mathrm{def}}{=} z:(y_1 y_2).\ \overline{y_i}.\ [\![\mathrm{Act}]\!]$$

Nat-Cell Revisited

- Suppose a cell can be empty. To raise "exception" in such a case, the cell becomes:

$$\mathrm{Empty}(a) \stackrel{\mathrm{def}}{=} a(z) :: z[?\underline{\mathrm{read}} : \underline{\mathrm{isempty}};\ \mathrm{Empty}(a)$$
$$?\underline{\mathrm{write}} : ?X;\ \mathrm{Cell}'(aX)\,]$$

$$\mathrm{Cell}'(aN) \stackrel{\mathrm{def}}{=} a(z) :: z[?\underline{\mathrm{read}} : !\underline{\mathrm{notempty}};\ !N;\ \mathrm{Cell}'(aN)$$
$$?\underline{\mathrm{write}} : ?X;\ \mathrm{Cell}'(aX)\,]$$

$$a^{+} : (\mathbf{1} \oplus \uparrow\mathrm{Nat}) \,\&\, (\downarrow\mathrm{Nat})$$

$$\mathrm{Reader}(\langle x \rangle P_1, P_2) \stackrel{\mathrm{def}}{=} \overline{a}(z) :: z!\underline{\mathrm{read}};\ [?\underline{\mathrm{isnotempty}};\ ?x;\ P_1$$
$$?\underline{\mathrm{isempty}} : P_2\,]$$

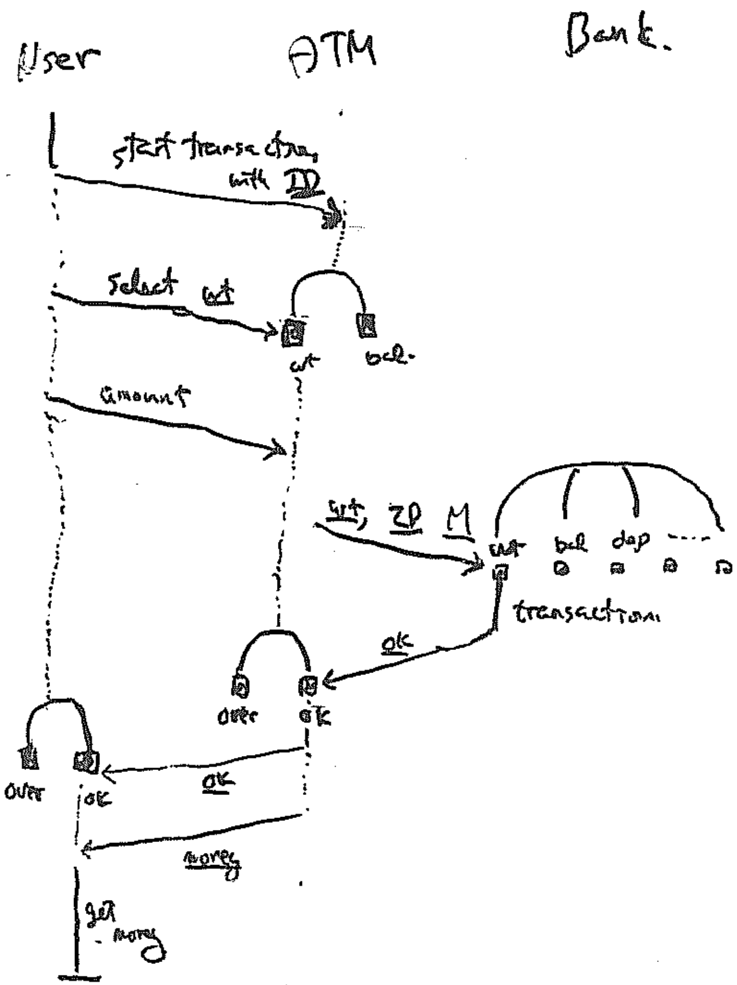
exception handling.

$$\mathrm{Writer}(aM) \stackrel{\mathrm{def}}{=} \overline{a}(z) :: z!\underline{\mathrm{write}};\ !M$$

$$a^{-} : (\mathbf{1} \,\&\, \downarrow\mathrm{Nat}) \oplus (\uparrow\mathrm{Nat})$$

Implementing ATM (1)

- Interaction pattern of ATM without recursion.



Implementing ATM (2)

$\mathrm{ATM}(ab) \stackrel{\mathrm{def}}{=}$

$[\text{illegible}]\langle z\rangle :: z?\mathrm{ID};\ [\,?\underline{wd};\ ?X;$ * ↔ user.

$\quad \overline{b}\langle w\rangle :: w!\underline{wt};\ !\mathrm{ID};\ !X;$ * ↔ bank.

$\qquad [\,?\underline{ok}:$ * ;

$\qquad\quad z!\underline{ok};\ !X;\ \mathrm{ATM}(ab),$ * ↔ user

$\qquad\ ?\underline{\mathrm{overdraft}}:$ * ↔ bank

$\qquad\quad z!\underline{over};\ \mathrm{ATM}(ab)\,],$ * ↔ user.

$\quad ?\underline{bal};$ * ↔ user

$\quad \overline{b}\langle w\rangle :: w!\underline{bal};\ ?X;$ * ↔ bank.

$\qquad z!X;\ \mathrm{ATM}(ab)\,]$

with user:

$$a^{+} : \underbrace{\downarrow \mathrm{Nat}}_{\text{userID}} \cdot \Big(\overbrace{(\uparrow \mathrm{Nat} \oplus \mathbb{1})}^{\text{withdraw}} \,\&\, \overbrace{\uparrow \mathrm{Nat}}^{\text{balance}} \Big)$$

with bank:

$$b^{-} : \underbrace{\big(\uparrow \mathrm{Code} \cdot \uparrow \mathrm{Nat} \cdot (\mathbb{1} \,\&\, \mathbb{1})\big)}_{\text{withdraw}} \oplus \underbrace{\big(\uparrow \mathrm{Code} \cdot \downarrow \mathrm{Nat}\big)}_{\text{balance}} \oplus \cdots$$

Implementing ATM (3)

$$\mathrm{ATM}(ab) \stackrel{\mathrm{def}}{=} a(z) :: z?\mathrm{ID}.\ \mathrm{Loop}(a, b, \mathrm{ID}, z)$$

$$
\begin{aligned}
\mathrm{Loop}(a, b, \mathrm{ID}, z) \stackrel{\mathrm{def}}{=} [\,& ?\underline{wd};\ ?X;\\
&\quad \overline{b}\langle w\rangle :: w!\underline{wd};\ !\mathrm{ID};\ !X;\\
&\quad\quad [\,?\underline{ok}:\\
&\quad\quad\quad z!\underline{ok};\ !X;\ \mathrm{Loop}(a, b, \mathrm{ID}, z)\\
&\quad\quad ?\underline{overdraft}:\\
&\quad\quad\quad z!\underline{over};\ \mathrm{Loop}(a, b, \mathrm{ID}, z)\,]\\
& ?\underline{bal};\\
&\quad \overline{b}\langle w\rangle :: w!\underline{bal};\ ?X;\\
&\quad\quad z!X;\ \mathrm{Loop}(a, b, \mathrm{ID}, z)\\
& ?\underline{exit};\ \mathrm{ATM}(ab)\,]
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{User}(a) \stackrel{\mathrm{def}}{=} \overline{a}\langle z\rangle :: z!3257;\ !\underline{bal};\ ?X;\\
!\underline{wt};\ !X;\ [\,?\underline{ok};\ ?Y;\ P\\
?\underline{over}:\ Q\,]
\end{aligned}
$$

* Look at the balance and withdraw all the money.

Implementing ATM (4)

- How can we know ATM and User have compatible behaviour?
- A closer look reveals ATM has a type (with user):

$$a^{+} : \downarrow\mathrm{Nat}\cdot[(\downarrow\mathrm{Nat}\cdot(\uparrow\mathrm{Nat}\cdot\odot \oplus \odot) \,\&\, (\uparrow\mathrm{Nat}\cdot\odot) \,\&\, \mathbb{1}]$$

or, more legibly:

$$a^{+} = \downarrow\mathrm{Nat}\cdot\alpha \quad \text{s.t.}\quad \underline{\alpha} = [(\downarrow\mathrm{Nat}\cdot(\uparrow\mathrm{Nat}\cdot\underline{\alpha} \oplus \underline{\alpha})) \,\&\, (\uparrow\mathrm{Nat}\cdot\underline{\alpha}) \,\&\, \mathbb{1}]$$

- Now User can be given a type:

$$a^{-} : \uparrow\mathrm{Nat}\cdot\alpha \quad \text{s.t.}\quad \underline{\alpha} = [(\uparrow\mathrm{Nat}\cdot(\downarrow\mathrm{Nat}\cdot\underline{\alpha} \,\&\, \underline{\alpha})) \oplus (\uparrow\mathrm{Nat}\cdot\underline{\alpha}) \oplus \mathbb{1}]$$

showing their compatibility.

...enting ATM (5)

...n combined with an appropriate bank

account, we have:

$$User(a) \mid b\nu(ATM(ab) \mid Bank(b))$$

$$\rightarrow \approx P \mid b\nu(ATM(ab) \mid \underbrace{Bank'(b)})$$

With the bank acc. of 3257 being Ø $.

which shows User always succeed

in withdrawing the money.

What We Hacked.

$l_1 l_2 \nu (! f' \nu f'' (\overline{e}f, f'a. (\overline{e}f. f'b.$

$ax. c\nu(\overline{z}c, cz. c\nu(zz_\nu(\overline{z}c', c'i. e\nu(\overline{l_2}e, ef.(\overline{f}a, ef.(\overline{f}b,$

$ef.(\overline{f}i, ef.\overline{f}z))))))$

$!f'' l_2 g.(\overline{g}f'', f''a.(\overline{g}f'', f''b.(\overline{g}f'', f''i.(\overline{g}f'', f''z.$

$c_1 c_2 c \nu(\overline{z}c, cy.(\overline{g}c, cy_2.\overline{y_1}c, cy_3.\overline{y_3}c,$

$c.(c_1.zy.e_0\nu(\overline{g}e_0, e_0 x.$

$e_1 w\nu(\overline{g}e_1, e_1 y.(\overline{g}w, e_1.wy.e_2\nu(\overline{g}e_2, e_2 y_1.$

$e_3\nu(\overline{g}e_3, e_3 y_2.e_4\nu(\overline{g}e_4, e_4.e_5\nu(\overline{w}e_5,$

$e_5 y.(\overline{g}i, e_5 y.(\overline{g}x, e_5.c_1' c_2'\nu(\overline{w}e_6, e_6 y.$

$(\overline{g}c_1', e_6 y.(\overline{g}c_2', e_6.zy.e_7\nu(\overline{g}e_7, e_7 y_1.$

$e_8\nu(\overline{g}e_8, e_8 y_1.e_9\nu(\overline{g}e_9, c_9.(c_1.e_{10}\nu(\overline{y_1}e_{10},$

$e_{10}, e_{10}.e_{11}\nu(\overline{z}e_{11}, e_{11} y.(\overline{y}x, e_{11}.e_0'\nu(\overline{l_1}e_0', e_0'z.(\overline{z}a, e_0'z.(\overline{z}b)))))$

$c_2'.e_{12}\nu(\overline{y_2}e_{12}, e_{12}.e_0'\nu(\overline{l_1}e_0', e_0'z.(\overline{z}a, e_0'z.(\overline{z}b)))))$

$c_2 e_{13}\nu \overline{g}e_{13}, e_{13} y.(\overline{g}w, e_{13}.wy.e_{14}\nu(\overline{g}e_{14},$

$e_{14}y_1.e_{15}\nu(\overline{g}e_{15}, e_{15}y_2.e_{16}\nu(\overline{g}e_{16}, e_{16}.e_{17}\nu(\overline{y_2}e_{17},$

$e_{17}\nu(\overline{w}y\,\overline{g}e_{17},$

$e_{17}x.e_{18}\nu(\overline{z}e_{18},$

$e_{18}y(\overline{g}x$

$e_{19}.e\nu(\overline{l_2}e,$

$ef.(\overline{f}a, ef.(\overline{f}b,$

$ef.(\overline{f}i, ef.\overline{f}z))))$

$c_3.e_0'\nu(\overline{l_1}e_0', e_0'z.(\overline{z}b))))))))$

5

Discussions.

Scope of π-hacking (1)

- Programming languages for concurrency:
  - → Design [Vasc92, PT93, ...]
  - → Types [Mil92, VH93, SP93, ...]
  - → Implementation [Turner 96, ...]

  cf. Tyco, Pict, Oz, ...

- "Core calculus" for semantic study.
  - → Encoding of λ-calculi [Mil90, Sang92, ...]
  - → Encoding of Proof Nets [Abr91, Bellin-Scott93, ..]
  - → Encoding of COOPS [Walker92]

- Study of behavioural equivalences.
  - * Closely related with the study of types, cf. [PS93, HY94, Yoshida96].
  - * Now a vast body of literature....

Scope of π-hacking (2)

- Basic study of expressiveness.
  - → Concurrent Combinators [HY94, RS95]
  - → Study on "summation" [NeP.96, Palam97]
  - → Asynchrony vs. Synchrony [HT91, HY93, Hon94,..]
- Connection to game semantics
  - * Making categories with name-passing processes.
  - * Hacking in π-calculus TOGETHER WITH domain theory and category theory, ... and more.

⟶ Towards 21st century.