A REFERENCE CELL IN T CALCULUS

\[
\text{Rep}(r,w,i) = (\forall e)(\overline{e}: | \text{ReadServer}(e,r) \\
| \text{WriteServer}(e,w))
\]

\[
\text{ReadServer}(e,r) = !e(c).d(v). (\exists v | \overline{e}v)
\]

\[
\text{WriteServer}(e,w) = !w(c,v'). d(v). (\exists | \overline{e}v')
\]

Example using reference cell:

\[
(\forall e) \overline{e} <c,v>. c. (\forall d) \overline{d}. d(e). @
\]

will receive the value v over the channel d
assuming no other processes interacting
with the reference cell.
new contents: \texttt{^Int}
run contents!\texttt{0}

def set \([v:\texttt{Int}\ c:\texttt{Sig}]\) =
    contents?\_ =
    (contents!\texttt{v} | c![])

def get \([\texttt{res}:/\texttt{Int}]\) =
    contents?\texttt{v} =
    (contents!\texttt{v} | \texttt{res!v})
Using Pact's Reference Cell

new done : ^[]
new res : ^Int

run ( set! [5 made (rchan done)]
     done? [] =
       (get! [(rchan res)]
         res? i = println! i))

5

def res i : Int = println! i
def done [] = $get! [res]
run set! [5 done]

5
Continuation-Passing Translation

\[ \text{run (def } f[x: \text{Int}, res: \text{Int}] = t! [x \times \text{res}] y! f) } \]

we create a channel \( f \), and we send it along \( y \).

\[ (\text{new } n: T \times! n) \]
creates a channel \( n \) and sends it along \( x \). It is the same as:

\[ x! (\text{new } n: T n) \]

Note it does not send the expression, it evaluates the expression and sends the value.
CPS - Continued

e.g. \[ x! [(\text{new } x : A x) \ (\text{new } y : B y)] \]

creates two new channels, packages into a single type (along with integer 23) and sends the result along \( x \).

**VALUE DECLARATIONS**

e.g. \( (\text{val } x = (\text{new } n : T [n \ n]) \ e) \)

binds \( x \) to the result of executing \( (\text{new } n : T [n \ n]) \) and then executes \( e \).

*Note: \( e \) blocks until the bindings in \( \text{val} \) expression have been established.*
**Application Syntax**

\[(v \, v_1 \, \ldots \, v_n)\]

e.g.

```scala
def double [s:String, r:(String)] = +$! [s <= r]
```

we can write \((\text{double}\, s)\) as a value dropping explicit result channel \(r\)

run print!(\text{double "soothe"})

\text{soothesoother}
**Functional Programming Style**

\[
\text{def } f[a_1:A_1 \ a_2:A_2 \ a_3:A_3 \ r:T] = r!\nu
\]

can be replaced by a "function definition" that avoids explicitly naming \( r \):

\[
\text{def } f(a_1:A_1 \ a_2:A_2 \ a_3:A_3):T = \nu
\]

**Anonymous Abstractions**

\[ \backslash a \]

is the same as:

\[
(\text{def } x \ a \ x)
\]
FOR LOOP EXAMPLE

def for [min:int max:int p: ![int /C]] d: !C]

  (def loop x:int =
    if (<x x max) then
      [new c: ^C]
        ( p! [x (rchan c)]
           1 c?[C] = loop! (+ x 1) )
    else
      done![]
      loop! min)

run (new done: ^C]

  (for! [1 4

      \[x c] = (print! x | c!C])
      (rchan done)

      1 done?[C] = print! "Done!"
   )

1 2 3 4 5 6 7
Done!
**Sequencing**

run
(val [] = (pr "hello")
(val [] = (pr "world")
())

equivalent to:

run ((pr "hello")
(pr "world")
())

**Semantics of Value Declarations & Sequencing**

\[
[(\text{val } p = v \ e)] = (\text{def } c \ p = [[e]]
\[v \rightarrow c]]
\]

\[v; \Rightarrow \text{val } [] = v\]
Using Reference Cell (with Apple Syntax & Sequencing)

run ((set 5);
   (prNL (int.toString (get)));
   (set 8);
   (prNL (int.toString (get)));
   ()))
type RepInt = [
  set = \[Int Sig\]
  get = \[/Int\]
]

def repInt (): RepInt =

  (new contents: ^Int
   run contents! 0
   [
     set = \[v:Int c:Sig\] =
       contents?_ = (contents! v । c! [ ])
     get = \[res:!Int\] =
       contents?v = (contents! v । res?v)
   ]
  )
REVISITED REFERENCE CELL USAGE

```scala
val ref1 = (ref1.int)
val ref2 = (ref2.int)
run ((ref2.set 5);
  (ref1.set 3);
  (prNL (int.toString (ref1.get)));
  (prNL (int.toString (ref2.get)));
  ())
```

```
3
5
```
LIS I_S

import "Std/List"
val l = (cons 6 (cons 7 (cons 8 nil)))
run println (car (cdr l))

FOLDING
(con 6 7 8 nil)
≡
(cons 6 (cons 7 (cons 8 nil)))

RIGHT FOLDING
(f < a a a2 ... an) ≡
(f (f (f a a1) a2) ... an)
Polymorphism

def print2nd[#:X l: (list X) p:/[X /String]]
  = if (null l) then print! "Null list"
    else if (null (cdr l)) then print! "Null tail"
    else print! (p (car (cdr l)))

The #: indicates it is a type parameter.

e.g.:

run print2nd![#:int (cons > 678 nil)
              int.toString ]

7

run print2nd![#:String (cons "one" "two" nil)
              \(s:String) = s ]

two
Abstract Types

val [ # Weekday
  m: Weekday t: Weekday w: Weekday
  r: Weekday f: Weekday s: Weekday
  n: Weekday
  sameDay: / [Weekday Weekday / Bool]
  tomorrow: / [Weekday / Weekday] ]

= [ # Int
  0 1 2 3 4 5 6
  \ (d1: Int d2: Int) = (== d1 d2)
  \ (d: Int) = (mod (+ d 1) 7)]

Now, we can use the abstract type, e.g.:

def weekend (d: Weekday): Bool =
  (11 (sameDay d s) (sameDay d n))
User-defined Type Constructors

Generic Reference Cell Constructor

\[
\begin{align*}
type \ (\text{Ref } x) &= [ \\
set &= 1[[x \ \text{Sig}]] \\
get &= 1[[1x]] \\
]
\end{align*}
\]

Ref is a parametric type -- it describes a family of types.

\[(\text{Ref } T) \equiv [\text{set} = 1[[T \ \text{Sig}]] \ \text{get} = 1[[1T]]]\]
def ref (#x init: x) : (Ref x) =
    (new contents: ^x
     run contents! init
    [
        set = $[v: x c: Sig] =
            contents?_ = (contents! v | c! [])
        get = $[res: x] =
            contents? v = (contents! v | res! v)
    ])

Generic Reference Cell Usage

val ref1 = (ref #Int 0)
val ref2 = (ref #String "one")

run ((ref1.set 5);
  (prNL (ref2.get));
  (prNL (int.toString (ref1.get)));
  ()
)

one
5

If the type parameter is omitted, PICT will infer it if possible.

e.g.  (ref #Int 0) ≜ (ref 0)
      (ref #String "one") ≜ (ref "one")