# A REFERENCE CELL IN π-CALCULUS

$$Ref(r,w,i) = (\nu l)(\,\bar{l}i \mid ReadServer(l,r) \mid WriteServer(l,w)\,)$$

$$ReadServer(l,r) = !\,r(c).\,l(v).\,(\,\bar{c}v \mid \bar{l}v\,)$$

$$WriteServer(l,w) = !\,w(c,v').\,l(v).\,(\,\bar{c} \mid \bar{l}v'\,)$$

Example using reference cell:

$$(\nu c)\,\bar{w}\langle c,v\rangle.\,c.\,(\nu d)\,\bar{r}d.\,d(e).\,Q$$

will receive the value $v$ over the channel $d$ assuming no other processes interacting with the reference cell.

# REFERENCE CELL IN PICT

```
new contents: ^Int
run contents!0
def set [v: Int c: Sig] =
    contents?_ =
    (contents!v | c![])
def get [res: /Int] =
    contents? v =
    (contents!v | res!v)
```

# Using Pict's Reference Cell

```
new done : ^[]
new res : ^Int
run ( set![5 (rchan done)]
      | done?[] =
          (get![(rchan res)]
          | res?i = printi!i))
```

5

---

```
def res i:Int = printi!i
def done [] = get![res]
run set![5 done]
```

5

# CONTINUATION-PASSING TRANSLATION

run (def f[x:Int res:/Int] = +![x x res]
    y!f)

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

(new n:T x!n)

creates a channel n and sends it along x. It is the same as:

x!(new n:T n)

Note it does not send the <u>expression</u>, it evaluates the expression and sends the value.

## CPS - Continued

e.g.

x![23 (new x:A x) (new y:B y)]

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

## VALUE DECLARATIONS

e.g.

(val x = (new n:T [n n]) e)

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

Note e **blocks** until the bindings in val expression have been established.

# APPLICATION SYNTAX

(v v₁ ... vₙ)

e.g.

```
def double[s:String r:(String]
    = +$![s s r]
```

we can write (double s) <u>as</u> <u>a value</u> dropping explicit result channel r

```
run print!(double "soothe")
```

soothesoothe

# FUNCTIONAL PROGRAMMING STYLE

def f[a1:A1 a2:A2 a3:A3 r: !T] = r!v

can be replaced by a "function definition" that avoids explizitly naming r:

def f(a1:A1 a2:A2 a3:A3):T = v

# ANONYMOUS ABSTRACTIONS

\a

is the same as:

(def x a x)

# FOR LOOP EXAMPLE

```
def for [min:Int max:Int f:![Int /[]] d:/[]]
  (def loop x:Int =
     if (<= x max) then
       (new c: ^[]
         (f![x (rchan c)]
         | c?[] = loop!(+ x 1) ))
     else
       d![]
   loop!min)

run (new done: ^[]
      (for![1 4
             \[x c] = (printi!x | c![])
             (rchan done)]
      | done?[] = print!"Done!") )
```

```
1
2
3
4
Done!
```

# SEQUENCING

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

equivalent to:

```
run ((pr "hello ");
     (pr "world");
     ()
    )
```

# SEMANTICS OF VALUE DECLARATIONS & SEQUENCING

$$[\![ (\text{val } p = v \ e) ]\!] = (\text{def } c \ p = [\![ e ]\!] \ [\![ v \to c ]\!])$$

$$v; \Rightarrow \text{val } [\,] = v$$

## USING REFERENCE CELL (WITH APP SYNTAX & SEQUENCING)

```
run ((set 5);
     (prNL (int.toString (get)));
     [redacted]
5    (set 8);
     (prNL (int.toString (get)));
     ())
```

# REFERENCE CELL IN PICT (REVISITED)

```
type RefInt = [
        set = /[Int Sig]
        get = /[/Int]
  ]

def refInt (): RefInt =
    (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

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

3
5

# LISTS

```
import "std/list"
val l = (cons 6 (cons 7 (cons 8 nil)))
run printi! (car (cdr l))
```

7

---

FOLDING

(cons > 6 7 8 nil)

$$\equiv$$

(cons 6 (cons 7 (cons 8 nil)))

---

(f > $a_1$ $a_2$ ... $a_n$ a)

$$\equiv$$

(f $a_1$ (f $a_2$ ... (f $a_n$ a)))

---

RIGHT FOLDING

(f < a $a_1$ $a_2$ ... $a_n$) $\equiv$

(f (f (f a $a_1$) $a_2$) ... $a_n$)

# 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 > 6 7 8 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 =
    (|| (sameday d s) (sameday d n))
```

# USER-DEFINED TYPE CONSTRUCTORS

## GENERIC REFERENCE CELL CONSTRUCTOR

```
type (Ref X) = [
        set = /[X Sig]
        get = /[/X]
    ]
```

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

(Ref T) ≡ [set=/[T Sig] get=/[/T]]

# GENERIC REFERENCE CELL IN PICT

```
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 ommitted, PICT will infer it if possible.

e.g. (ref #Int 0) ≡ (ref 0)

(ref #String "one") ≡ (ref "one")