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

```plaintext
new done : ^[]
new res : ^Int

run ( set!(5 and! (rchan done))
  | done?[] =
  | (get![(rchan res)]
  | res?i = print!(i))

5
```

def res i: Int = print!(i)
def done [] = &get! [res]
run set! [5 done]

5
Continuation-Passing Transilation

\[
\text{run} \left( \text{def } f[x:\text{Int} \, \text{res}:/\text{Int}] = +![x \times \text{res}] \right) \]

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

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

\[
\text{\(x! \left[23 \ (\text{new } x:A \ x) \ (\text{new } y:B \ y)\right]\)}
\]

creates two new channels, packages into a single tuple (along with integers 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**

\[(v_1 v_2 ... v_n)\]

e.g.

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

We can write (double `s`) as a value dropping explicit result channel `r` run `print!(double "soothe")`

soothesoothe
**Functional Programming Style**

```haskell
def f[a1:A1 a2:A2 a3:A3 r:1T] = r
```

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

```haskell
def f(a1:A1 a2:A2 a3:A3):T = r
```

**Anonymous Abstractions**

\[
\lambda a
\]

is the same as:

```lisp
(def x a x)
```
FOR LOOP EXAMPLE

```python
def for C[min:int max:int i:[[int /C]] d:/C]
    (def loop x:int =
        if (< x max) then
            [new c : ^[]
            ( f! [x (rchan c)]
            | c?[] = loop! (+ x 1) )]
        else
            [new done: ^[]
            loop! min ]
    run (new done: ^[]
        (for! [1 4
            \[x c] = (print! x | c!c]
        (rchan done)]
        | done?[] = print! "Done!"
    )
```
Sequencing

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

equivalent to:

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

Semantics of Value Declarations & Sequencing

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

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

```clojure
(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 !Ø
    [
      set = \[v:Int c:Sig] =
        contents? v = (contents! v | c ![])
      get = \[res:!!Int] =
        contents? v = (contents! v | res!! v)
    ]
)
REVISITED REFERENCE CELL USAGE

val ref1 = (ref1:int)
val ref2 = (ref2:int)
run ((ref2.set 5);
    (ref1.set 3);
    (prNL (int.to_string (ref1.get)));
    (prNL (int.to_string (ref2.get)));
    ()))
import "std/lib";  
val l = (cons 6 (cons 7 (cons 8 nil)));  
run print! (car (cdr l));

7

FOLDING
(cons > 6 7 8 nil)
≡
(cons 6 (cons 7 (cons 8 nil)));

(f > a, a2, ..., an, a)
≡
(f a, (f a2, ..., (f an, a)));

RIGHT FOLDING
(f < 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 > 6 7 8 nil)
  int.to.String ]

> 

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
same day : /[Weekday Weekday /Book]
tomorrow : /[Weekday /Weekday]]

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

\[
\text{type } (\text{Ref } x) = [
\text{set} = / [x \text{ Sig}] \\
\text{get} = / [1x]
]
\]

\text{Ref} is a parametric type -- it describes a family of types.

\((\text{Ref } T) \equiv [\text{set}=/ [T \text{ Sig}] \text{ get}=/ [1T]]\)
def ref (#x init: x) : (Ref x) =

  (new contents: ^x
    run contents! init
   
    set = \[v: x c: Sig]\n        contents?_ = (contents! v | c! [])
    get = \[res: / x]\n        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")
**Nomadic Pict**

**Declarations**

\[ \text{type } T = T' \]  
\[ \text{new } c : T \ P \]  
\[ \text{agent } a = P \text{ and } \ldots \text{ and } a' = P' \text{ in } Q \]  
\[ \text{agent creation} \]  
\[ \text{migrate to } s P \]  
\[ \text{agent migration} \]  
\[ \text{def } f[\ldots] = P \text{ and } \ldots \text{ and } f'[\ldots] = P' \text{ in } Q \]  
\[ \text{process abstraction} \]

**Processes**

\[ (P \ P Q) \]  
\[ (D P) \]  
\[ () \]  
\[ \text{parallel composition} \]  
\[ \text{local declaration} \]  
\[ \text{null process} \]
Communication

\( c! v \) \hspace{2cm} \text{output } v \text{ on channel } c \text{ in current agent }
\( c? p = p \) \hspace{2cm} \text{input}
\( c?#p = p \) \hspace{2cm} \text{replicated input}
\( \text{if } v \text{ then } P \text{ else } Q \) \hspace{2cm} \text{conditional}

\( \text{if-local } \langle a \rangle c! v \text{ then } P \text{ else } Q \) \hspace{2cm} \text{test-and-send to agent a on this site.}
\( \langle a \rangle c! v \) \hspace{2cm} \text{send to agent a on this site.}
\( \langle a\text{es} \rangle c! v \) \hspace{2cm} \text{send to agent a on site } s.
\( \text{wait } c? p = p \text{ timeout } t \rightarrow Q \) \hspace{2cm} \text{input with timeout (secs)}
\( \text{terminate} \) \hspace{2cm} \text{kill agent}

\( c\text{ea}! v \) \hspace{2cm} \text{location-independent output to channel c at agent a.}
Nomadic Pict Example

getApplet?*{a s} \rightarrow
agent b =
migrate to s \rightarrow
(aes)ack!b!b
in 0

Location-independent Communication

\langle ae? \rangle c!v

c@a!v
Mobile Agent Example

new answer: "String

def spawn [s:Site prompt: String] =
  (agent b =
    (migrate to s
     answer@a! (sys.read prompt))
   in
     ()
     (spawn! [s1 "How are you?"]
      spawn! [s2 "when do we start?"]
      answer? & s = print! s

..."

This code (part of agent a) spawns two agents at sites s1 and s2, and prints answers coming on its "answer" channel.
type ReplInt =
[
  set = / [Agent int sig]
  get = / [Agent /int]
]
def replInt [s: Site r: /ReplInt] =
  (new set: ^ [Agent int sig]
   new get: ^ [Agent /int]
   agent replIntAg =
   (new contents: ^ Int
    run contents! 0
    migrate to s
    (set ?!! [a: Agent v: int c: sig] =
     contents?_. = (contents!v | e!c)
     1 get?? [a: Agent res: /int] =
     contents? v = (contents!v | res@a!v))
  )
)r! [set = \ [a: Agent v: int c: sig] =
  set @ replIntAg ![a v c]
get = \ [a: Agent res: /int] = get @ replIntAg (a and)
Nomadic Net Reference Cell Usage

val cell1 = (repInt 51)
val cell2 = (repInt 52)

agent a =
  (
    (cell2.set a 3);
    (prNL (int.toString (cell1.get a)));
    (prNL (int.toString (cell2.get a)));
    ()
  )