A Few Small Orc Programs

Jayadev Misra

Department of Computer Science
University of Texas at Austin

http://orc.csres.utexas.edu

ETH, Zurich
March 7, 2014

A Orc program

- An experiment tosses two dice.
  Experiment is a success if the dice throws sum to 7.

- $exp(n)$ runs n experiments and reports the number of successes.

```
def toss() = Random(6) + 1
 -- toss returns a random number between 1 and 6

def exp(0) = 0
def exp(n) = exp(n - 1)
               + (if toss() + toss() = 7 then 1 else 0)
```

Translation of the dice throw program

```
def  toss() =  add(x, 1)  <x< Random(6)
def  exp(n) =
   ( Ift(b)  ≫ 0
    | Iff(b)  ≫
      ( add(x, y)
          <x< ( exp(m)  <m< sub(n, 1) )
          <y< ( Ift(bb)  ≫ 1 | Iff(bb)  ≫ 0 )
             <bb< equals(p, 7)
                <p< add(q, r)
                   <q< toss()
                   <r< toss()
      )

   )  <b< equals(n, 0)
```

Note: $2n$ parallel calls to *toss*().

Orc Calculus

- External sites:
    - A site is called like a procedure with parameters.
    - Site returns any number of values.
    - The value is published.
- Combinators
- Definitions

No notion of data type, thread, process, channel, synchronization, parallelism $\cdots$

Orc Language

- Orc Calculus
- Syntactic Sweeteners
  - Data Types: Number, Boolean, String, with Java operators
  - Conditional Expression: *if* E *then* F *else* G
  - Data structures: Tuple, List, Record
  - Pattern Matching; Clausal Definition
  - Closure
  - Class for active objects
- Site Library

Every Orc language program is translated to Orc calculus.

Orc Calculus: Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f > x > g$	Sequential composition

$f \mid g$: Evaluate f and g independently. Publish all values from both.

$f > x > g$:
For all values published by f do g. Publish only the values from g.

Orc Calculus: Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f > x > g$	Sequential composition

$f \mid g$: Evaluate f and g independently. Publish all values from both.

$f > x > g$:
For all values published by f do g. Publish only the values from g.

Orc Calculus: Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f > x > g$	Sequential composition

$f \mid g$: Evaluate f and g independently. Publish all values from both.

$f > x > g$:
For all values published by f do g. Publish only the values from g.

Orc Calculus: Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f > x > g$	Sequential composition

$f \mid g$: Evaluate f and g independently. Publish all values from both.

$f > x > g$:
For all values published by f do g. Publish only the values from g.

Schematic of Sequential composition

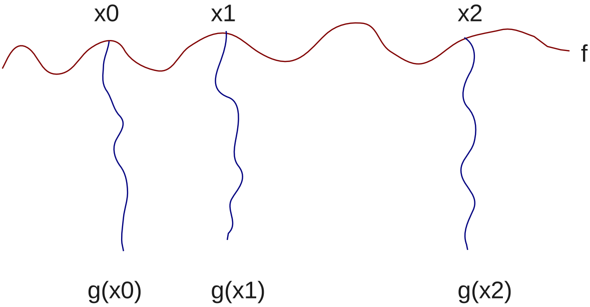


Figure: Schematic of $f >x> g$

Sequential composition: $f\ >x>\ g$

For all values published by f do g.
Publish only the values from g.

- $CNN(d)\ >x>\ Email(address, x)$
  - Call $CNN(d)$.
  - Bind result (if any) to x.
  - Call $Email(address, x)$.
  - Publish the value, if any, returned by $Email$.

- $(CNN(d) \mid BBC(d))\ >x>\ Email(address, x)$
  - May call $Email$ twice.
  - Publishes up to two values from $Email$.

Notation: $f \gg g$ for $f\ >x>\ g$, if x is unused in g.

Right Associative: $f\ >x>\ g\ >y>\ h$ is $f\ >x>\ (g\ >y>\ h)$

Subset Sum

Given integer n and list of integers xs.

$parsum(n, xs)$ publishes all sublists of xs that sum to n.

`parsum(5,[1,2,1,2]) = [1,2,2], [2,1,2]`

`parsum(5,[1,2,1])` is silent

```
def $parsum(0, [\,]) = [\,]$

def $parsum(n, [\,]) = stop$

def $parsum(n, x : xs) =$
    $parsum(n, xs)$                        $--$ all sublists that do not include $x$
  | $parsum(n - x, xs)$ $>ys>$ $x : ys$    $--$ all sublists that include $x$
```

Subset Sum

Given integer n and list of integers xs.

$parsum(n, xs)$ publishes all sublists of xs that sum to n.

`parsum(5,[1,2,1,2]) = [1,2,2], [2,1,2]`

`parsum(5,[1,2,1])` is silent

$$def\ \ parsum(0, [\,]) = [\,]$$

$$def\ \ parsum(n, [\,]) = stop$$

$$\begin{array}{ll} def\ \ parsum(n, x : xs) = & \\ \quad parsum(n, xs) & --\ \text{all sublists that do not include } x \\ \quad |\ parsum(n - x, xs)\ >ys>\ x : ys & --\ \text{all sublists that include } x \end{array}$$

Subset Sum

Given integer n and list of integers xs.

$parsum(n, xs)$ publishes all sublists of xs that sum to n.

`parsum(5,[1,2,1,2]) = [1,2,2], [2,1,2]`

`parsum(5,[1,2,1])` is silent

$$def\ \ parsum(0, [\,]) = [\,]$$

$$def\ \ parsum(n, [\,]) = stop$$

$$\begin{aligned} &def\ \ parsum(n, x : xs) = \\ &\qquad parsum(n, xs) && --\ \text{all sublists that do not include } x \\ &\quad\ |\ parsum(n - x, xs)\ \ {>}ys{>}\ x : ys && --\ \text{all sublists that include } x \end{aligned}$$

Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f > x > g$	Sequential composition
→ if f halts without publishing do g	$f \; ; \; g$	Otherwise

Subset Sum (Contd.), Backtracking

Given integer n and list of integers xs.

$seqsum(n, xs)$ publishes the first sublist of xs that sums to n.

“First” is smallest by index, lexicographically.
`seqsum(5,[1,2,1,2]) = [1,2,2]`

`seqsum(5,[1,2,1])` is silent

$$\mathit{def}\ \ seqsum(0, [\,]) = [\,]$$

$$\mathit{def}\ \ seqsum(n, [\,]) = \mathit{stop}$$

$$\begin{aligned} &\mathit{def}\ \ seqsum(n, x : xs) = \\ &\quad x : seqsum(n - x, xs) \\ &\;\; ;\ seqsum(n, xs) \end{aligned}$$

Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f >x> g$	Sequential composition
→for some x from g do f	$f <x< g$	Pruning
if f halts without publishing do g	$f\ ;\ g$	Otherwise

Pruning: $f \;<x<\; g$

For some value published by g do f.

- Evaluate f and g in parallel.
  - Site calls that need x are suspended.
    Consider $(M() \mid N(x)) \;<x<\; g$
- When g returns a (first) value:
  - Bind the value to x.
  - Kill g.
  - Resume suspended calls.
- Values published by f are the values of $(f \;<x<\; g)$.

Notation: $f \ll g$ for $f \;<x<\; g$, if x is unused in f.

Left Associative: $f \;<x<\; g \;<y<\; h$ is $(f \;<x<\; g) \;<y<\; h$

Note: Concurrent computation of f, g and h, above.

Subset Sum (Contd.), Concurrent Backtracking

Publish the first sublist of xs that sums to n.

Run the searches concurrently.

$$def\ \ parseqsum(0, [\,]) \;=\; [\,]$$

$$def\ \ parseqsum(n, [\,]) \;=\; stop$$

$$\begin{array}{l} def\ \ parseqsum(n, x : xs) \;= \\ \quad (p \;;\; q) \\ \qquad <p<\; x : parseqsum(n - x, xs) \\ \qquad <q<\; parseqsum(n, xs) \end{array}$$

Note: Neither search in the last clause may succeed.

val; a syntactic sweetener

Write $f \ {<}x{<}\ g$ as

$$\begin{array}{l} val \ \ x = \ g \\ \qquad\ f \end{array}$$

Deflation

- Given expression $C(\ldots, e, ..)$, single value expected at e.
  Translated to $C(\ldots, x, ..)$ $<x<$ e where x is fresh.

- Applicable hierarchically.

  $(1|2) * (10|100)$ is translated to

  $(Times(x, y)$ $<x<$ $(1 \mid 2))$ $<y<$ $(10 \mid 100)$, or without parentheses
  $Times(x, y)$ $<x<$ $(1 \mid 2)$ $<y<$ $(10 \mid 100)$

- Implication:
  Arguments of site calls are evaluated in parallel.
  Note: A strict site is called when all arguments have been evaluated.

Deflation

- Given expression $C(\ldots, e, ..)$, single value expected at e.
  Translated to $C(\ldots, x, ..)$ $<x<$ e where x is fresh.

- Applicable hierarchically.

  $(1|2) * (10|100)$ is translated to

  $(Times(x, y)$ $<x<$ $(1 \mid 2))$ $<y<$ $(10 \mid 100)$, or without parentheses
  $Times(x, y)$ $<x<$ $(1 \mid 2)$ $<y<$ $(10 \mid 100)$

- Implication:
  Arguments of site calls are evaluated in parallel.
  Note: A strict site is called when all arguments have been evaluated.

Deflation

- Given expression $C(\ldots, e, ..)$, single value expected at e.
  Translated to $C(\ldots, x, ..)$ $<x<$ e where x is fresh.

- Applicable hierarchically.

  $(1|2) * (10|100)$ is translated to

  $(Times(x, y)$ $<x<$ $(1 \mid 2))$ $<y<$ $(10 \mid 100)$, or without parentheses
  $Times(x, y)$ $<x<$ $(1 \mid 2)$ $<y<$ $(10 \mid 100)$

- Implication:
  Arguments of site calls are evaluated in parallel.
  Note: A strict site is called when all arguments have been evaluated.

Parsing using Recursive Descent

Consider the grammar:

$$\begin{array}{lcl} \mathit{expr} & ::= & \mathit{term} \mid \mathit{term} + \mathit{expr} \\ \mathit{term} & ::= & \mathit{factor} \mid \mathit{factor} * \mathit{term} \\ \mathit{factor} & ::= & \mathit{literal} \mid (\mathit{expr}) \\ \mathit{literal} & ::= & 3 \mid 5 \end{array}$$

Parsing strategy

For each non-terminal, say *expr*, define *expr*(*xs*) for string *xs*: publish all suffixes of *xs* such that the prefix is a *expr*.

```
def isexpr(xs) = expr(xs) >[]> true ; false
```

To avoid multiple publications (in ambiguous grammars),

```
def isexpr(xs) =
   val res = expr(xs) >[]> true ; false
    res
```

```
−− Test

isexpr
([”(”,”(”,”3”,” ∗ ”,”3”,”)”,”)”,” + ”,”(”,”3”,” + ”,”3”,”)”])
       −− ((3*3))+(3+3)


:: true
```

Site for each non-terminal

$$
\begin{array}{lll}
\text{Given:} & \mathit{expr} ::= & \mathit{term} \mid \mathit{term} + \mathit{expr} \\
\text{Rewrite:} & \mathit{expr} ::= & \mathit{term}\ (\epsilon \mid\ + \mathit{expr})
\end{array}
$$

$$
\begin{array}{lcl}
\text{def}\ \ \mathit{expr}(xs) & = & \mathit{term}(xs)\ \ {>}ys{>}\ (ys \mid ys\ \ > \text{"+"}\ \ : zs{>}\ \mathit{expr}(zs)) \\[1ex]
\text{def}\ \ \mathit{term}(xs) & = & \mathit{factor}(xs)\ \ {>}ys{>}\ (ys \mid ys\ \ > \text{"*"}\ \ : zs{>}\ \mathit{term}(zs)) \\[1ex]
\text{def}\ \ \mathit{factor}(xs) & = & \ \ \mathit{literal}(xs) \\
 & & \mid xs\ \ > \text{"("}\ \ : ys{>}\ \mathit{expr}(ys)\ \ > \text{")"}\ \ : zs{>}\ zs \\[1ex]
\text{def}\ \ \mathit{literal}(n : xs) & = & n\ \ > \text{"3"} > xs \mid n\ \ > \text{"5"} > xs \\
\text{def}\ \ \mathit{literal}([\,]) & = & \mathit{stop}
\end{array}
$$

Parallel or

Expressions f and g return single booleans. Compute the parallel or.

$\mathit{Ift}(b)$, $\mathit{Iff}(b)$: boolean b,
Returns a signal if b is true/false; remains silent otherwise.

$$\begin{array}{l} \mathit{val}\ x = f \\ \mathit{val}\ y = g \end{array}$$

$$\mathit{Ift}(x) \gg \mathit{true} \mid \mathit{Ift}(y) \gg \mathit{true} \mid (x \,||\, y)$$

Parallel or

Expressions f and g return single booleans. Compute the parallel or.

$Ift(b)$, $Iff(b)$: boolean b,
Returns a signal if b is true/false; remains silent otherwise.

$$\begin{array}{l} val\ x = f \\ val\ y = g \end{array}$$

$$Ift(x) \gg true \mid Ift(y) \gg true \mid (x \,||\, y)$$

Parallel or; contd.

Compute the parallel or and return just one value:

$$\begin{array}{l}
\mathit{val}\ \ x = f \\
\mathit{val}\ \ y = g \\
\mathit{val}\ \ z = \mathit{Ift}(x) \gg \mathit{true} \mid \mathit{Ift}(y) \gg \mathit{true} \mid (x \,||\, y) \\
\quad z
\end{array}$$

But this continues execution of g if f first returns true.

$$\begin{array}{l}
\mathit{val}\ \ z = \\
\quad \mathit{val}\ \ x = f \\
\quad \mathit{val}\ \ y = g \\
\\
\quad \mathit{Ift}(x) \gg \mathit{true} \mid \mathit{Ift}(y) \gg \mathit{true} \mid (x \,||\, y) \\
z
\end{array}$$

Parallel or; contd.

Compute the parallel or and return just one value:

$$\begin{array}{l}
val\ \ x = f \\
val\ \ y = g \\
val\ \ z = Ift(x) \gg true \mid Ift(y) \gg true \mid (x \,||\, y) \\
\quad z
\end{array}$$

But this continues execution of g if f first returns true.

$$\begin{array}{l}
val\ \ z = \\
\quad val\ \ x = f \\
\quad val\ \ y = g \\
\\
\quad Ift(x) \gg true \mid Ift(y) \gg true \mid (x \,||\, y) \\
z
\end{array}$$

Mutable Store: Some Factory Sites

`Ref(n)`	Mutable reference with initial value `n`
`Array(n)`	Array of size `n` of Refs
`Semaphore(n)`	Semaphore with initial value n
`Channel()`	Unbounded (asynchronous) channel
`Table(n,f)`	Array of size `n` of immutable values of `f`

$\mathit{Ref}(3)\ >r>\ r.\mathit{write}(5) \gg r.\mathit{read}()$, or $\mathit{Ref}(3)\ >r>\ r := 5 \gg r?$

$\mathit{Array}(3)\ >a>\ a(0) := \mathit{true} \gg a(1)?$

$\mathit{Semaphore}(1)\ >s>\ s.\mathit{acquire}() \gg \mathit{Println}(0) \gg s.\mathit{release}()$

$\mathit{Channel}()\ >ch>\ (ch.\mathit{get}() \mid ch.\mathit{put}(3) \gg \mathit{stop})$

$\mathit{val}\ \ ch = \mathit{Table}(10, \mathit{lambda}(\_) = \mathit{Channel}())$

Exception Handling

Client calls site `server` to request service.
The server “may” request authentication information.

```
def  request(x) =
   val  exc =  Channel()   −− returns a channel site

      server(x, exc)
    | exc.get()  >r> exc.put(auth(r))  ≫  stop
```

Packet Reassembly Using Sequence Numbers

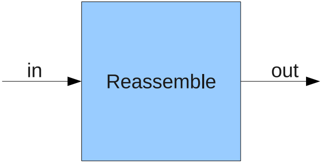


Figure: Packet Reassembler

- Packet with sequence number i is at position p_i in the input channel.
- Given: $|i - p_i| \leq k$, for some positive integer k.
- Then $p_i \leq i + k \leq p_{i+2\times k}$. Let $d = 2 \times k$.

Packet Reassembly Program

$$\mathit{def}\ \ \mathit{reassembly}(\mathit{read}, \mathit{write}, d) = \qquad -- \text{ d must be positive}$$

$$\quad \mathit{val}\ \ \mathit{ch} = \ \mathit{Table}(d, \mathit{lambda}(\_) = \mathit{Channel}())$$

$$\quad \mathit{def}\ \ \mathit{input}() = \ \mathit{read}()\ \ {>}(n, v){>}\ \mathit{ch}(n\%d).\mathit{put}(v)\ \ \gg \mathit{input}()$$

$$\quad \mathit{def}\ \ \mathit{output}(i) = \ \mathit{ch}(i).\mathit{get}()\ \ {>}v{>}\ \mathit{write}(v)\ \ \gg \mathit{output}((i+1)\%d)$$

$$\mathit{input}()\ |\ \mathit{output}(0) \qquad -- \text{ Goal expression}$$

Note: $n\%d$ is n mod d.

Depth-first search of undirected graph
Recursion over Mutable Structure

N: Number of nodes in the graph.

$conn$: $conn(i)$ the list of neighbors of node i, $0 \leq i < N$

$parent$: Mutable array of length N.
$parent(i) = v$, $v \geq 0$, means v is the parent node of i
$parent(i) < 0$ means parent of i is yet to be determined

Once i has a parent, it continues to have that parent.

Start Depth-first search from node 0.
$parent(0) = N$

Invariant

$dfs(i, xs)$: starts a depth-first search from all nodes in xs in order,

i already has a parent or $i = N$.

$xs \subseteq conn(i)$, i.e., xs is some set of neighbors of i.

All neighbors of i not in xs already have parents.

Depth-first search

```
val N = 6                      -- N is the number of nodes in the graph
val parent = Table(N, lambda(_) = Ref(−1) )

def dfs(_, [ ]) = signal

def dfs(i, x : xs) =

   if (parent(x)? ≥ 0) then dfs(i, xs)

   else parent(x) := i ≫ dfs(x, conn(x)) ≫ dfs(i, xs)

dfs(N, [0])                    -- start depth-first search from node 0
```

Quicksort

- In situ permutation of an array.
- Array segments are simultaneously sorted.
- Partition of an array segment proceed from left and right simultaneously.
- Combine Concurrency, Recursion, and Mutable Data Structures.

Traditional approaches

- Pure functional programs do not admit in-situ permutation.
- Imperative programs do not highlight concurrency.
- Typical concurrency constructs do not combine well with recursion.

Program Structure

- array a to be sorted.
- $segmentsort(u, v)$ sorts the segment $a(u)..a(v-1)$ in place and publishes a signal.
- To sort the whole array: $segmentsort(0, a.length?)$

Program Structure; Contd.

- $part(p, s, t)$ partitions segment (s, t) with element p. Publishes m where:

  left subsegment: $a(i) \leq p$ for all i, $s \leq i \leq m$, and
  right subsegment: $a(i) > p$, for all i, $m < i < t$.

- Assume $a(s)? \leq p$, so the left subsegment is non-empty.

```
def swap(i,j) = (i?,j?) >(x,y)> (i := y, j := x) ≫ signal
def quicksort(a) =
    def segmentsort(u,v) =
        if v − u > 1 then
            part(a(u)?,u,v) >m>
            swap(a(u),a(m)) ≫
            (segmentsort(u,m), segmentsort(m+1,v)) ≫ signal
        else signal
segmentsort(0, a.length?)
```

Partition segment (s, t) with element p, given $a(s) \leq p$

- $lr(i)$ publishes the index of the leftmost item in the segment that exceeds p; publishes t if no such item.
- $rl(i)$ publishes the index of the rightmost item that is less than or equal to p. Since $a(s) \leq p$, item exists.

$$def\ \ lr(i) = \ Ift(i <: t) \gg Ift(a(i)? \leq p) \gg lr(i+1)\ ;\ i$$

$$def\ \ rl(i) = \ Ift(a(i)? :> p) \gg rl(i-1)\ ;\ i$$

Goal Expression of $part(p, s, t)$:

$$\begin{array}{l} (lr(s+1), rl(t-1))\ \ >(s', t')> \\ (if\ \ (s' < t')\ then\ \ swap(a(s'), a(t')) \gg part(p, s', t') \\ \ else\ \ t') \end{array}$$

Putting the Pieces together: Quicksort

```
def swap(i, j) = (i?, j?) >(x, y)> (i := y, j := x) ≫ signal

def quicksort(a) =
      def segmentsort(u, v) =
         def part(p, s, t) =
            def lr(i) = Ift(i < t) ≫ Ift(a(i)? ≤ p) ≫ lr(i + 1) ; i
            def rl(i) = Ift(a(i)? :> p) ≫ rl(i − 1) ; i                          #

            (lr(s + 1), rl(t − 1)) >(s', t')>
            (if (s' < t') then swap(a(s'), a(t')) ≫ part(p, s', t')
             else t')                                                           #

         if v − u > 1 then
            part(a(u)?, u, v) >m>
            swap(a(u), a(m)) ≫
            (segmentsort(u, m), segmentsort(m + 1, v)) ≫ signal
         else signal
segmentsort(0, a.length?)
```

Class: Pure Rendezvous

```
def class pairSync() =
    val s = Semaphore(0)
    val t = Semaphore(0)

    def put() = s.acquire() ≫ t.release()
    def get() = s.release() ≫ t.acquire()
stop
```

Rendezvous with Data Transfer

```
def  class zeroChannel() =
     val  s =  Semaphore(0)
     val  w =  BoundedChannel(1)

     def  put(x) =  s.acquire()  ≫  w.put(x)
     def  get() =   s.release()  ≫  w.get()
stop
```

Class: Readers-Writers

- Readers and Writers need access to a shared file.
- Any number of readers may read the file simultaneously.
- A writer needs exclusive access.

Readers-Writers API

- A reader calls $start(true)$, writer $start(false)$ to gain access.
- The system (class) returns a signal to grant access.
- Both readers and writers call $end()$ on completion of access.
- $start(\cdots)$ is blocking, $end()$ non-blocking.

Implementation Strategy

- Each call to *start* is queued with the id of the caller.

- A *manager* loops forever, maintaining the invariant:
  There is no active writer (no writer has been granted access).
  Number of active readers = *na.value*, where *na* is a counter.

- On each iteration, *manager* picks the next queue entry.
  If a reader: grant access and increment *na*.
  If a writer:
    wait until all readers complete (*na*'s value = 0),
    grant access to writer,
    wait until the writer completes.

Implementation Strategy; Callback

- The id assigned to a caller is a new semaphore.
- A request is (b, s): b boolean, s semaphore.
  $b = \mathit{true}$ for reader, $b = \mathit{false}$ for writer,
  each caller waits on $s.\mathit{acquire}()$
- The manager grants a request by executing $s.\mathit{release}()$

Reader-Writer; Call API

```
val req = Channel()
val na = Counter()

def startread() =
    val s = Semaphore(0)
    req.put((true, s)) ≫ s.acquire()

def startwrite() =
    val s = Semaphore(0)
    req.put((false, s)) ≫ s.acquire()

def endread() = na.dec()

def endwrite() = na.dec()
```

Reader-Writer; Main Loop

$$def \;\; manager() = grant(req.get()) \gg manager()$$

$$def \;\; grant((true, s)) = na.inc() \gg s.release() \quad --\text{ Reader}$$

$$\begin{aligned} &def \;\; grant((false, s)) = \quad --\text{ Writer} \\ &\quad na.onZero() \gg na.inc() \gg s.release() \gg na.onZero() \end{aligned}$$

Putting the pieces together: Reader-Writer

```
def class readerWriter1() =
    val req = Channel()     val na = Counter()

    def startread() =          val s = Semaphore(0)
    req.put((true, s)) ≫ s.acquire()

    def startwrite() =         val s = Semaphore(0)
    req.put((false, s)) ≫ s.acquire()

    def endread() = na.dec()
    def endwrite() = na.dec()

    def grant((true, s)) = na.inc() ≫ s.release()    −− Reader

    def grant((false, s)) =      −− Writer
    na.onZero() ≫ na.inc() ≫ s.release() ≫ na.onZero()

    def manager() = grant(req.get()) ≫ manager()

manager()
```

Callback using one semaphore each for Readers and Writers

```
def class readerWriter2() =
    val req = Channel()
    val na = Counter()
    val (r, w) = (Semaphore(0), Semaphore(0))

    def startread() = req.put(true) ≫ r.acquire()
    def startwrite() = req.put(false) ≫ w.acquire()

    def endread() = na.dec()
    def endwrite() = na.dec()

    def grant(true) = na.inc() ≫ r.release()    −− Reader

    def grant(false) =     −− Writer
      na.onZero() ≫ na.inc() ≫ w.release() ≫ na.onZero()

    def manager() = grant(req.get()) ≫ manager()

manager()
```

Reader-Writer; dispense with the queue

- Dispense with the queue.
  Introduce a class that keeps nr and nw, counts of readers and writers.

- Calling $put(true/false)$ increments the appropriate count.

- Calling $get()$ decrements a count and returns $true/false$.

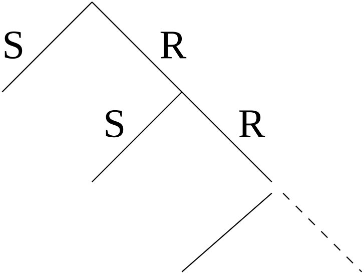
- Simulate fairness for get, as in removing from a channel.
  If $nr? > 0$, $nr?$ is eventually decremented.
  If $nw? > 0$, $nw?$ is eventually decremented.
  Use coin toss to simulate fairness.

Real time: Metronome

External site $Rwait(t)$ returns a signal after t time units.

$metronome$ publishes a $signal$ every time unit.

$$def\ \ metronome() = \underbrace{signal}_{S}\ \mid\ (\underbrace{Rwait(1) \gg metronome()}_{R})$$



Unending string of Random digits

$$metronome() \gg Random(10) \quad --\text{ one every unit}$$

$$def\ \ rand\_seq(dd) = \qquad --\text{ at a specified rate}$$

$$Random(10) \mid Rwait(dd) \gg rand\_seq(dd)$$

A time-based class; Stopwatch

- A stopwatch allows the following operations:

  $start()$: (re)starts and publishes a signal

  $halt()$: stops and publishes current value

- Other operations: $reset()$ and $isrunning()$.

Application: Measure running time of a site

```
def class profile(f) =
  val sw = Stopwatch()

  def runningtime() = sw.start() ≫ f() ≫ sw.halt()

  stop

-- Usage
def burntime() = Rwait(100)

profile(burntime).runningtime()
```

Response Time Game

- Show a random digit, v, for 3 secs.
- Then print an unending sequence of random digits.
- The user presses a key when he thinks he sees v.
- Output $(true, response\ time)$, or $(false, \_)$ if v has not appeared. Then end the game.

Response Game: Program

```
val sw = Stopwatch()
val (id, dd) = (3000, 100)   -- initial delay, digit delay
def rand_seq() =      -- Publish a random sequence of digits
  Random(10) | Rwait(dd) ≫ rand_seq()
def game() =
  val v = Random(10)   -- v is the seed for one game
  val (b, w) =
    Rwait(id) ≫ sw.reset() ≫ rand_seq() >x> Println(x) ≫
    Ift(x = v) ≫ sw.start() ≫ stop
   | Prompt( "Press ENTER for SEED "+v ) ≫
    sw.isrunning() >b> sw.pause() >w> (b, w)

if b then   -- Goal expression of game()
  ( "Your response time = " + w + " milliseconds." )
else ( "You jumped the gun." )
game()
```

Shortest Path Algorithm with Lights and Mirrors

- Source node sends rays of light to each neighbor.
- Edge weight is the time for the ray to traverse the edge.
- When a node receives its first ray, sends rays to all neighbors. Ignores subsequent rays.
- Shortest path length = time for sink to receive its first ray. Shortest path length to node i = time for i to receive its first ray.

Graph structure in $Succ()$

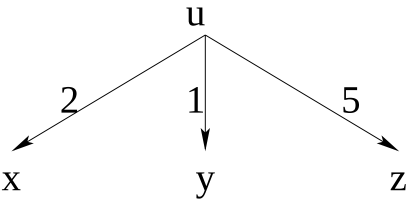


Figure: Graph Structure

$Succ(u)$ publishes $(x, 2)$, $(y, 1)$, $(z, 5)$.

Algorithm

```
def $eval(u,t)$ =    record value $t$ for $u$ $\gg$
                     for every successor $v$ with $d$ = length of $(u,v)$ :
                       wait for $d$ time units $\gg$
                       $eval(v, t+d)$

Goal :               $eval(source, 0)$ |
                     read the value recorded for the $sink$
```

Record path lengths for node u in FIFO channel u.

Algorithm(contd.)

```
$def\ \ eval(u,t) =$    record value $t$ for $u$ $\gg$
                         for every successor $v$ with $d$ = length of $(u,v)$ :
                        wait for $d$ time units $\gg$
                        $eval(v, t+d)$

$Goal:$                 $eval(source, 0)$ |
                        read the value recorded for the $sink$
```

A cell for each node where the shortest path length is stored.

```
$def\ \ eval(u,t) =$    $u := t$ $\gg$
                        $Succ(u)\ \ >(v,d)>$
                        $Rwait(d)$ $\gg$
                        $eval(v, t+d)$

{-  $Goal:$ -}          $eval(source, 0)$ | $sink?$
```

Algorithm(contd.)

```
def eval(u, t) =    u := t ≫
                    Succ(u) >(v, d)>
                    Rwait(d) ≫
                    eval(v, t + d)

{- Goal :-}         eval(source, 0) | sink?
```

- Any call to *eval*(u, t): Length of a path from source to u is t.
- First call to *eval*(u, t): Length of the shortest path from source to u is t.
- *eval* does not publish.

Drawbacks of this algorithm

- Running time proportional to shortest path length.
- Executions of *Succ*, *put* and *get* should take no time.