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.

\[
\begin{align*}
def \text{toss()} & = \text{Random}(6) + 1 \\
& \quad \text{—— toss returns a random number between 1 and 6}
\end{align*}
\]

\[
\begin{align*}
def \text{exp}(0) & = 0 \\
def \text{exp}(n) & = \text{exp}(n - 1) \\
& \quad + (\text{if toss()} + \text{toss()} = 7 \text{ then } 1 \text{ else } 0)
\end{align*}
\]
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()
        )
    )
)

<bb< 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 · · ·
Orc Language

- Orc Calculus

- Syntactic Sweeteners
  - Data Types: Number, Boolean, String, with Java operators
  - Conditional Expression: $\textit{if } E \textit{ then } F \textit{ else } G$
  - 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:
  
  \[
  \text{do } f \text{ and } g \text{ in parallel} \quad f \mid g \quad \text{Symmetric composition}
  \]
  
  \[
  \text{for all } x \text{ from } f \text{ do } g \quad f \triangleright x \triangleright g \quad \text{Sequential composition}
  \]

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

  \( f \triangleright x \triangleright g \):
  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:

  \[
  \text{do } f \text{ and } g \text{ in parallel } \quad f \mid g
  \]
  Symmetric composition

  \[
  \text{for all } x \text{ from } f \text{ do } g \quad f >x> g
  \]
  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, $\textit{CNN}(d)$
  Publishes the value returned by the site.

- **Composition** of two Orc expressions:

  do $f$ and $g$ in parallel $\quad f \parallel g$  
  Symmetric composition

  for all $x$ from $f$ do $g$ $\quad f > x > g$  
  Sequential composition

$f \parallel g$: 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, $\text{CNN}(d)$
  Publishes the value returned by the site.

- **Composition** of two Orc expressions:

  - `do f and g in parallel`  \( f \bowtie g \)  Symmetric composition
  - `for all x from f do g`  \( f \blacktriangleright x \blacktriangleright g \)  Sequential composition

\( f \bowtie g \): Evaluate \( f \) and \( g \) independently. Publish all values from both.

\( f \blacktriangleright x \blacktriangleright g \): For all values published by \( f \) do \( g \). Publish only the values from \( g \).
Schematic of Sequential composition

Figure: Schematic of $f \circ x \circ g$
Sequential composition: \( f \gg x \gg g \)

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

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

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

Notation: \( f \gg g \) for \( f \gg x \gg g \), if \( x \) is unused in \( g \).

Right Associative: \( f \gg x \gg g \gg y \gg h \) is \( f \gg x \gg (g \gg y \gg h) \)
Subset Sum

Given integer $n$ and list of integers $xs$.

$\text{parsum}(n, xs)$ publishes all sublists of $xs$ that sum to $n$.

$\text{parsum}(5, [1, 2, 1, 2]) = [1, 2, 2], [2, 1, 2]$  
$\text{parsum}(5, [1, 2, 1])$ is silent

\[
def \text{parsum}(0, []) = []
\]

\[
def \text{parsum}(n, []) = \text{stop}
\]

\[
def \text{parsum}(n, x : xs) =
\]
\[
| \text{parsum}(n, xs) \quad \text{--- all sublists that do not include } x
\]
\[
| \text{parsum}(n - x, xs) > ys > x : ys \quad \text{--- all sublists that include } x
\]
Subset Sum

Given integer \( n \) and list of integers \( xs \).

\( \text{parsum}(n, xs) \) publishes all sublists of \( xs \) that sum to \( n \).

\[ \text{parsum}(5, [1, 2, 1, 2]) = [1, 2, 2], [2, 1, 2] \]

\( \text{parsum}(5, [1, 2, 1]) \) is silent

\[
\begin{align*}
\text{def } \text{parsum}(0, []) &= [] \\
\text{def } \text{parsum}(n, []) &= \text{stop} \\
\text{def } \text{parsum}(n, x : xs) &= \\
&\phantom{{}\text{parsum}(n, xs)} \quad \text{all sublists that do not include } x \\
&\phantom{{}\text{parsum}(n, xs)} \quad | \quad \text{parsum}(n - x, xs) > ys > x : ys \quad \text{all sublists that include } x
\end{align*}
\]
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

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

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

\[
def parsum(n, x : xs) =
parsum(n, xs) \quad \text{--- all sublists that do not include } x
\]

\[
| \quad parsum(n - x, xs) \quad >ys> x : ys \quad \text{--- all sublists that include } x
\]
Structure of Orc Expression

- **Simple**: just a site call, \( \text{CNN}(d) \)
- **Composition** of two Orc expressions:

\[
\begin{align*}
\text{do } f \text{ and } g \text{ in parallel} & \quad f \parallel g \quad \text{Symmetric composition} \\
\text{for all } x \text{ from } f \text{ do } g & \quad f > x > g \quad \text{Sequential composition} \\
\rightarrow \text{ if } f \text{ halts without publishing do } g & \quad f ; g \quad \text{Otherwise}
\end{align*}
\]
Subset Sum (Contd.), Backtracking

Given integer \( n \) and list of integers \( xs \).

\( \text{seqsum}(n, xs) \) publishes the first sublist of \( xs \) that sums to \( n \).

“First” is smallest by index, lexicographically.

\( \text{seqsum}(5, [1, 2, 1, 2]) = [1, 2, 2] \)

\( \text{seqsum}(5, [1, 2, 1]) \) is silent

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

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

\[
\text{def } \text{seqsum}(n, x : xs) = \\
\quad x : \text{seqsum}(n - x, xs) \;
\text{seqsum}(n, xs)
\]
Structure of Orc Expression

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

  - $f$ and $g$ in parallel: $f \mid g$  
    Symmetric composition
  
  - for all $x$ from $f$ do $g$: $f > x > g$  
    Sequential composition
  
  - $\rightarrow$ for some $x$ from $g$ do $f$: $f < x < g$  
    Pruning
  
  - if $f$ halts without publishing do $g$: $f ; g$  
    Otherwise
Pruning: \( f \prec x \prec g \)

For some value published by \( g \) do \( f \).

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

Notation: \( f \ll g \) for \( f \prec x \prec g \), if \( x \) is unused in \( f \).

Left Associative: \( f \prec x \prec g \prec y \prec h \) is \( (f \prec x \prec g) \prec y \prec h \)

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.

$$\text{def } parseqsum(0, []) = []$$

$$\text{def } parseqsum(n, []) = \text{stop}$$

$$\text{def } parseqsum(n, x : xs) =$$

$$(p ; q)$$

$$<p< x : parseqsum(n - x, xs)$$

$$<q< parseqsum(n, xs)$$

Note: Neither search in the last clause may succeed.
val; a syntactic sweetener

Write $f < x < g$ as

\[
\text{val } x = g \\
\quad f
\]
Deflation

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

• Applicable hierarchically.

  $(1|2) \ast (10|100)$ 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 \( C(..., e, ..) \), single value expected at \( e \).
  Translated to \( C(..., x, ..) <x< e \) where \( x \) is fresh.

• Applicable hierarchically.

\[
(1\mid2) \ast (10\mid100) \text{ is translated to }
\]

\[
(Times(x, y) <x< (1 \mid 2)) <y< (10 \mid 100), \text{ or without parentheses }
Times(x, y) <x< (1 \mid 2) <y< (10 \mid 100)
\]

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

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

- Applicable hierarchically.

  
  $$(1|2) \ast (10|100)$$

  is translated to

  $$(Times(x, y) < x < (1 | 2)) < y < (10 | 100),$$

  or without parentheses

  $$Times(x, y) < x < (1 | 2) < y < (10 | 100)$$

- 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:

\[
\begin{align*}
expr & ::= \ term \mid \ term + expr \\
\term & ::= \ factor \mid \ factor \ast \ term \\
\factor & ::= \ literal \mid (expr) \\
\literal & ::= 3 \mid 5
\end{align*}
\]
Parsing strategy

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

\[
def \textit{isexpr}(xs) = \textit{expr}(xs) >[ \ ]> true ; false
\]

To avoid multiple publications (in ambiguous grammars),

\[
def \textit{isexpr}(xs) = \\
val res = \textit{expr}(xs) >[ \ ]> true ; false \\
res
\]

\[\text{—— Test}\\
\textit{isexpr} \\
([” , ” , ”3”, ” * ” , ”3”, ” )”, ” + ” , ” (” , ”3”, ” + ” , ”3”, ” )” ])] \\
\quad —— ((3*3))+(3+3)
\]

\[:: true\]
Site for each non-terminal

Given: \( expr ::= term \mid term + expr \)

Rewrite: \( expr ::= term (\epsilon \mid + expr) \)

def \( expr(xs) \) = \( term(xs) >ys> (ys \mid ys > ”+” : zs> expr(zs)) \)
def \( term(xs) \) = \( factor(xs) >ys> (ys \mid ys > ”*” : zs> term(zs)) \)
def \( factor(xs) \) = \( literal(xs) \)
\[ xs > ”(” : ys> expr(ys) > ”)” : zs> zs \]
def \( literal(n : xs) \) = \( n > ”3” > xs \mid n > ”5” > xs \)
def \( literal([]) \) = \( stop \)
Parallel or

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

\[ \text{Ift}(b), \text{Iff}(b): \text{boolean} \ b, \]

Returns a signal if \( b \) is true/false; remains silent otherwise.

\[
\begin{align*}
\text{val} & \ x = f \\
\text{val} & \ y = g \\
\text{Ift}(x) \gg \text{true} & \mid \text{Ift}(y) \gg \text{true} & \mid (x \parallel y)
\end{align*}
\]
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.

$$
\text{val } x = f \\
\text{val } y = g
$$

$Ift(x) \gg true \mid Ift(y) \gg true \mid (x \mid\mid y)$
Parallel or; contd.

Compute the parallel or and return just one value:

\[
\begin{align*}
\text{val } x &= \text{f} \\
\text{val } y &= \text{g} \\
\text{val } z &= \text{Ift}(x) \gg true \mid \text{Ift}(y) \gg true \mid (x \parallel y) \\
& \quad z
\end{align*}
\]

But this continues execution of \text{g} if \text{f} first returns true.

\[
\begin{align*}
\text{val } z &= \\
& \quad \text{val } x = \text{f} \\
& \quad \text{val } y = \text{g} \\
& \quad \text{Ift}(x) \gg true \mid \text{Ift}(y) \gg true \mid (x \parallel y) \\
& \quad z
\end{align*}
\]
Parallel or; contd.

Compute the parallel or and return just one value:

```
val x = f
val y = g
val z = Ift(x) \gg true \lor Ift(y) \gg true \lor (x \parallel y)
```

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

```
val z =
val x = f
val y = g

Ift(x) \gg true \lor Ift(y) \gg true \lor (x \parallel y)
```

```
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 \( f \)

\[
\text{Ref}(3) \succ r \succ r.write(5) \Rightarrow r.read(), \text{ or } \text{Ref}(3) \succ r \succ r := 5 \Rightarrow r?
\]
\[
\text{Array}(3) \succ a \succ a(0) := \text{true} \Rightarrow a(1)?
\]
\[
\text{Semaphore}(1) \succ s \succ s.acquire() \Rightarrow \text{Println}(0) \Rightarrow s.release()
\]
\[
\text{Channel()} \succ ch \succ (\text{ch.get()} | \text{ch.put}(3) \Rightarrow \text{stop})
\]
\[
\text{val ch = Table}(10, \text{lambda}(\_ = \text{Channel}())
\]
Exception Handling

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

```python
def request(x):
    val exc = Channel()  # returns a channel

    server(x, exc)
    | exc.get() >r> exc.put(auth(r)) >> stop
```
Packet Reassembly Using Sequence Numbers

- 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 $d = 2 \times k$. 

Figure: Packet Reassembler
Packet Reassembly Program

```python
def reassembly(read, write, d) =  --- d must be positive

val ch = Table(d, lambda(_ = Channel()))

def input() = read() > (n, v) > ch(n%d).put(v) >> input()

def output(i) = ch(i).get() > v > write(v) >> output((i + 1)%d)

input() | output(0)  --- 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))

\[
\begin{align*}
def\ dfs(_, []) &= \text{signal} \\
def\ dfs(i, x : xs) &= \\
\quad \text{if } (parent(x)? \geq 0) \text{ then } dfs(i, xs) \\
\quad \text{else } \ &\quad parent(x) := i \ \triangleright \ dfs(x, conn(x)) \ \triangleright \ dfs(i, xs) \\
\end{align*}
\]

dfs(\( N \), [0])  
\( \quad \) 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.

- $\text{segmentsort}(u, v)$ sorts the segment $a(u)\ldots a(v - 1)$ in place and publishes a signal.

- To sort the whole array: $\text{segmentsort}(0, a.length)$
Program Structure; Contd.

- \( \text{part}(p, s, t) \) 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.

```python
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.

\[
def lr(i) = \text{Ift}(i < : t) \gg \text{Ift}(a(i)? \leq p) \gg lr(i + 1) ; i
\]

\[
def rl(i) = \text{Ift}(a(i)? : > p) \gg rl(i - 1) ; i
\]

Goal Expression of \(\text{part}(p, s, t)\):

\[
(lr(s + 1), rl(t - 1)) > (s', t') > \\
(\text{if } (s' < t') \text{ then } \text{swap}(a(s'), a(t'))) \gg \text{part}(p, s', t') \\
\text{else } t')
\]
Putting the Pieces together: Quicksort

```python
def swap(i, j) = (i?, j?) > (x, y) > (i := y, j := x) \text{ signal}
def quicksort(a) =
    def segmentsort(u, v) =
        def part(p, s, t) =
            def lr(i) = Ift(i < t) \text{ Ift(a(i)? \leq p) lr(i + 1); i}
            def rl(i) = Ift(a(i)? :> p) \text{ rl(i - 1); i}

            (lr(s + 1), rl(t - 1)) > (s', t') >
            (if (s' < t') then swap(a(s'), a(t')) \text{ 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)) \text{ signal}
        else signal
    segmentsort(0, a.length?)
```
def class pairSync() =
  val s = Semaphore(0)
  val t = Semaphore(0)

  def put() = s.acquire() \(\gg\) t.release()
  def get() = s.release() \(\gg\) 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 \texttt{start(true)}, writer \texttt{start(false)} to gain access.
- The system (class) returns a signal to grant access.
- Both readers and writers call \texttt{end()} on completion of access.
- \texttt{start(\ldots)} is blocking, \texttt{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. 
  $b = true$ for reader, $b = false$ for writer, 
  each caller waits on $s.acquire()$

- The manager grants a request by executing $s.release()$
Reader-Writer; Call API

```scala
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

\[
def \text{manager}() = \text{grant}(\text{req}.\text{get}()) \gg \text{manager}()
\]

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

\[
def \text{grant}((\text{false}, s)) = \quad -- \quad \text{Writer}
\text{na}.\text{onZero}() \gg \text{na}.\text{inc}() \gg \text{s}.\text{release}() \gg \text{na}.\text{onZero}()
\]
Putting the pieces together: Reader-Writer

```python
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

```python
def class readerWriter2() =
    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 \( \text{put}(\text{true}/\text{false}) \) increments the appropriate count.

- Calling \( \text{get}() \) decrements a count and returns \( \text{true}/\text{false} \).

- Simulate fairness for \( \text{get} \), 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 $R_{wait}(t)$ returns a signal after $t$ time units. $metronome$ publishes a $signal$ every time unit.

$$\text{def } metronome() = \text{signal}_{S} | (R_{wait}(1) \gg metronome())_{R}$$
Unending string of Random digits

\[ \text{metronome}() \Rightarrow \text{Random}(10) \quad -- \quad \text{one every unit} \]

\[ \text{def} \quad \text{rand\_seq}(dd) = \quad -- \quad \text{at a specified rate} \]

\[ \text{Random}(10) | \text{Rwait}(dd) \Rightarrow \text{rand\_seq}(dd) \]
A time-based class; Stopwatch

- A stopwatch allows the following operations:
  
  \textit{start}(): (re)starts and publishes a signal
  
  \textit{halt}(): stops and publishes current value

- Other operations: \textit{reset}() and \textit{isrunning}().
Application: Measure running time of a site

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

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

stop

— Usage

def burnttime() = Rwait(100)

profile(burntime).runningtime()```

48
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 \((true, \text{response time})\), or \((false, _)\) if \( v \) has not appeared. Then end the game.
Response Game: Program

```python
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) >>
        if (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 $\text{Succ}(u)$

$\text{Succ}(u)$ publishes $(x, 2)$, $(y, 1)$, $(z, 5)$.

Figure: Graph Structure
Algorithm

\[\text{def } \text{eval}(u, t) = \quad \text{record value } t \text{ for } u \quad \Rightarrow\]
\[\text{for every successor } v \text{ with } d = \text{length of } (u, v) :\]
\[\text{wait for } d \text{ time units } \Rightarrow\]
\[\text{eval}(v, t + d)\]

Goal:
\[\text{eval}(\text{source}, 0) |\]
\[\text{read the value recorded for the sink}\]

Record path lengths for node \( u \) in FIFO channel \( u \).
Algorithm (contd.)

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

Goal:

- \( \text{eval}(\text{source}, 0) \) |
- read the value recorded for the \text{sink}

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

```python
def eval(u, t) =
    u := t 
    \( \text{Succ}(u) \) \( > (v, d) > \)
    \( \text{Rwait}(d) \) 
    \( \text{eval}(v, t + d) \)
```

{ – Goal :– } \( \text{eval}(\text{source}, 0) \) | \text{sink}?
Algorithm (contd.)

\[
\text{def } \text{eval}(u, t) = u := t \gg \\
\text{Succ}(u) > (v, d) > \\
\text{Rwait}(d) \gg \\
\text{eval}(v, t + d)
\]

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

- Any call to \text{eval}(u, t): Length of a path from source to \textit{u} is \textit{t}.
- First call to \text{eval}(u, t): Length of the shortest path from source to \textit{u} is \textit{t}.
- \text{eval} does not publish.
Drawbacks of this algorithm

- Running time proportional to shortest path length.
- Executions of $\textit{Succ}$, $\textit{put}$ and $\textit{get}$ should take no time.