Structured Wide-Area Programming: Orc Programming Examples

Jayadev Misra

Department of Computer Science
University of Texas at Austin

http://orc.csres.utexas.edu
Some Algorithms

- Enumeration and Backtracking
- Using Closures
- List Fold, Map-reduce
- Parsing using Recursive Descent
- Exception Handling
- Process Network
- Quicksort
- Graph Algorithms: Depth-first search, Shortest Path
Enumeration

Given: integer \( n \), list of integers \( x_s \)
Return all subsequences of \( x_s \) that sum to \( n \).

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

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

\[
def \text{sum}(_, []) = \text{stop}
\]

\[
def \text{sum}(n, x : xs) =
\begin{align*}
\text{sum}(n - x, xs) > &ys > x : ys \\
| \text{sum}(n, xs)
\end{align*}
\]
Backtracking: Use of Otherwise

Given: integer $n$, list of integers $xs$
Return the “first” subsequence of $xs$ that sums to $n$.

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

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

\[
\text{def } \text{sum}(\_, [[]]) = \text{stop}
\]

\[
\text{def } \text{sum}(n, x : xs) = x : \text{sum}(n - x, xs) ; \text{sum}(n, xs)
\]
Backtracking: Eight queens

Place 8 queens on a chessboard so that no queen captures another.

Figure: Backtrack Search for Eight queens
Eight queens; contd.

- $xs$: partial placement of queens (list of values from 0..7)
- $extend(xs)$ publishes all solutions that are extensions of $xs$.
- $open(xs)$ publishes the columns that are open in the next row.
- Solve the original problem by calling $extend([])$.

```python
def extend(xs) =
    if (length(xs) == 8) then xs
    else (open(xs) ≫ j ≫ extend(j : xs))
```
Using Closure

A UNITY Program

\[ x, y = 0, 0 \]

\[ x < y \rightarrow x := x + 1 \]
\[ y := y + 1 \]

- Program has: variable declarations
  - a set of functions

- Variables are initialized as given.

- Program is run by: choosing a function arbitrarily, choosing functions fairly.
Corresponding Orc program

\[
\text{val } (x, y) = (\text{Ref}(0), \text{Ref}(0))
\]

\[
\text{def } f1() = \text{Ift}(x? <: y?) \gg x := x? + 1
\]

\[
\text{def } f2() = y := y? + 1
\]

Run the program by:

- choosing a function arbitrarily,
- choosing functions fairly.
def unity(fs) =
  val arlen = length(fs)
  val fnarray = Array(arlen)

{ -- populate() transfers from list fs to array fnarray -- }
def populate(_, []) = signal
def populate(i, g : gs) = fnarray(i) := g \triangleright populate(i + 1, gs)

{ -- Execute a random statement and loop. Randomness guarantees fairness. -- }
def exec() = random(arlen) \triangleright j \triangleright fnarray(j)?() \triangleright exec()

{ -- Initiate the work -- }
populate(0, fs) \triangleright exec()
Running the example program

\begin{align*}
\text{val } (x, y) &= (\text{Ref}(0), \text{Ref}(0)) \\
\text{def } f1() &= \text{Ift}(x \prec y) \quad \Rightarrow \quad x := x + 1 \\
\text{def } f2() &= y := y + 1 \\
\text{unity([f1,f2])}
\end{align*}
Associative Fold

• Define $a\text{fold}(f, x_s)$ where $f$ is an associative binary function and $x_s$ is a non-empty list.

• Goal is to combine elements in parallel.

• Each iteration reduces adjacent pairs of items to single values.

• Iterations continue until there is a single value.
Associative Fold; contd.

\[
\begin{align*}
def\, a\text{fold}(f, [x]) &= x \\
def\, a\text{fold}(f, xs) &= \\
\end{align*}
\]

\[
\begin{align*}
def\, step([]) &= [] \\
def\, step([x]) &= [x] \\
def\, step(x : y : xs) &= f(x, y) : step(xs) \\
\end{align*}
\]

\[
\begin{align*}
\text{a\text{fold}(f, step(xs))}
\end{align*}
\]

- \(f(x, y) : step(xs)\) is an implicit fork-join.
- \(f(x, y)\) executes concurrently with \(step(xs)\).
- All calls to \(f\) execute concurrently within each iteration of \(a\text{fold}\).
Associative and Commutative Fold

- Transfer list items to a channel (arbitrary order of items).
- Fold any two channel items and put the result in the channel.

```python
def acfold(f, xs):
    val c = Channel()

def xfer([]) = stop
def xfer(x : xs) = (c.put(x), xfer(xs))

def combine(1) = c.get()
def combine(m) =
    c.get() >>= x >> c.get() >> y >>
    (c.put(f(x, y)) >>= stop | combine(m - 1))

xfer(xs) | combine(length(xs))
```
• Given is a list of tasks.

• A processor from a processor pool is assigned to process a task. Each task may be processed independently, yielding a result.

• If a processor does not respond within time $T$, a new processor is assigned to the task.

• After all the results have been computed, the results are reduced by calling $reduce$. 
Implementation

- **processlist** processes a list of tasks concurrently. 
  \(\text{process}(t)\) processes a single task \(t\).
  \(\text{process}(t)\) publishes a result; **processlist** a list of results.

- Function **process** first acquires a processor. 
  It assigns the task to the processor. 
  If the processor responds within time \(T\), it publishes the result. 
  Else, it repeats these steps.

- **process\((t)\)** may never complete if the processors keep failing.

- The list of published results are reduced by function **reduce**.
map-reduce

def processlist([]) = []
def processlist(t : ts) = process(t) : processlist(ts)

def process(t) =
  val processor = Processorpool()
  val (result, b) = (processor(t), true) | (Rwait(T), false)
  if b then result else process(t)

processlist(tasks) >x> reduce(x)
Consider the grammar:

\[
expr ::= term \mid term + expr
\]

\[
term ::= factor \mid factor \ast term
\]

\[
factor ::= literal \mid (expr)
\]

\[
literal ::= 3 \mid 5
\]
Parsing strategy

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

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

To avoid multiple publications (in ambiguous grammars),

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

--------- Test

```python
isexpr
([""", "", "", "3", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "", "`,

:: true
Function for each non-terminal

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

\[
\text{def } expr(xs) = term(xs) >ys> (ys \mid ys > "+" : zs> expr(zs))
\]
\[
\text{def } term(xs) = factor(xs) >ys> (ys \mid ys > "+" : zs> term(zs))
\]
\[
\text{def } factor(xs) = literal(xs) \\
\mid xs > "(" : ys> expr(ys) > ")" : zs> zs
\]
\[
\text{def } literal(n : xs) = n > "3" > xs \mid n > "5" > xs
\]
\[
\text{def } literal([]) = stop
\]
Exception Handling

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

\[
\text{def } \text{request}(x) = \\
\text{val } exc = \text{Channel()} \text{ -- returns a channel site}
\]

\[
\text{server}(x, exc) \\
\mid exc.\text{get()} > r > exc.\text{put}(\text{auth}(r)) \gg \text{stop}
\]
A process network consists of: processes and channels.

The processes run autonomously, and communicate via the channels.

A network is a process; thus hierarchical structure. A network may be defined recursively.

A channel may have intricate communication protocol.

Network structure may be dynamic, by adding/deleting processes/channels during its execution.
Channels

- For channel \( c \), treat \( c\text{.put} \) and \( c\text{.get} \) as site calls.

- In our examples, \( c\text{.get} \) is blocking and \( c\text{.put} \) is non-blocking.

- We consider only FIFO channels. Other kinds of channels can be programmed as sites. We show rendezvous-based communication later.
Typical Iterative Process

Forever: Read \( x \) from channel \( c \), compute with \( x \), output result on \( e \):

\[
def \ p(c,e) = c.get() \quad > x > \quad \text{Compute}(x) \quad > y > \quad e.put(y) \quad \gg \ p(c,e)
\]

Figure: Iterative Process
Composing Processes into a Network

Process (network) to read from both \( c \) and \( d \) and write on \( e \):

\[
def \text{net}(c, d, e) = p(c, e) \mid p(d, e)
\]

Figure: Network of Iterative Processes
Workload Balancing

Read from \( c \), assign work randomly to one of the processes.

\[
\text{def bal}(c, c', d') = c \cdot \text{get}() > x > \text{random}(2) > t >
\]
\[
(\text{if } t = 0 \text{ then } c' \cdot \text{put}(x) \text{ else } d' \cdot \text{put}(x)) \gg
\]
\[
\text{bal}(c, c', d')
\]

\[
\text{def workbal}(c, e) = \text{val } c' = \text{Channel}()
\]
\[
\text{val } d' = \text{Channel}()
\]
\[
\text{bal}(c, c', d') \mid \text{net}(c', d', e)
\]
Deterministic Load Balancing

- Retain input order in the output.
- `distr` alternatively copies input to `c'` and `c''`.
- `coll` alternatively copies from `d'` and `d''` to output.

Diagram:

```
  c  -----> distr ----> p(c',d') ----> d' ----> d
     |         |                  |         |
     v         v                  v         v
  c'' -------+-----------------------+--------
             |                         |        
             |                         |        
             v                         v        
      p(c'',d'') ----> coll -----> d'' ----> d
```

Equations:

- `p(c',d')`
- `p(c'',d'')`
Deterministic Load Balancing

```python
def detbal(in, out) =

def distributor(c, c', c'') =
c.get() > x > c'.put(x) >
c.get() > y > c''.put(y) >
distributor(c, c', c'')

def collector(d', d'', d) =
d'.get() > x > d.put(x) >
d''.get() > y > d.put(y) >
collector(d', d'', d)

val (in', in'') = (Channel(), Channel())
val (out', out'') = (Channel(), Channel())

\[
\text{distributor}(in, in', in'') \mid \text{collector}(out', out'', out) \\
\mid p(in', out') \mid p(in'', out'')
\]
```
Deterministic Load Balancing with $2^n$ servers

Construct the network recursively.

\[
\text{recBal}(0,c,d) \\
\text{distr} ightarrow \\
c' \rightarrow \text{recBal}(n-1,c',d') \\
\text{coll} ightarrow \\
\text{distr} \rightarrow \\
c'' \rightarrow \text{recBal}(n-1,c'',d'')}
\]
Recursive Load Balancing Network

\[
\text{def } \text{recbal}(0, \text{in}, \text{out}) = P(\text{in}, \text{out})
\]

\[
\text{def } \text{recbal}(n, \text{in}, \text{out}) = \\
\text{def } \text{distributor}(c, c', c'') = \cdots
\]

\[
\text{def } \text{collector}(d', d'', d) = \cdots
\]

\[
\text{val } (\text{in}', \text{in}'') = (\text{Channel}(), \text{Channel}())
\]
\[
\text{val } (\text{out}', \text{out}'') = (\text{Channel}(), \text{Channel}())
\]

\[
\text{distributor}(\text{in}, \text{in}', \text{in}'') \mid \text{collector}(\text{out}', \text{out}'', \text{out}) \\
\mid \text{recbal}(n - 1, \text{in}', \text{out}') \mid \text{recbal}(n - 1, \text{in}'', \text{out}'')
\]
An Iterative Process: Transducer

Compute \( f(x) \) for each \( x \) in channel \( in \) and output to \( out \), in order.

```python
def transducer(in, out, fn) =
in.get() \>\> out.put(fn(x)) \>\> transducer(in, out, fn)
```
Pipeline network

Apply function \( f \) to each input: \( f(x) = h(g(x)) \), for some \( g \) and \( h \).

\[
def\ pipe(in, out, g, h) =
  val\ c = Channel()
  transducer(in, c, g) | transducer(c, out, h)
\]

\[
\begin{array}{ccc}
\text{in} & \rightarrow & g \\
\text{c} & \rightarrow & h \\
\text{out} & \rightarrow &
\end{array}
\]
Recursive Pipeline network

Consider computing factorial of each input.

\[
fac(x) = \begin{cases} 
1 & \text{if } x = 0 \\
x \times fac(x - 1) & \text{if } x > 0 
\end{cases}
\]

Suppose \( x \leq N \), for some given \( N \).
Outline of a program

```python
def fac(N, in, out) =
    val (in', out') = (Channel(), Channel())
    front(in, out, in', out') | fac(N - 1, in', out')
```

Diagram:
- Fac(N-1)
- Front
- Fac(N)
Implementation of $\text{Fac}_0$

- receive input $x$, $x = 0$
- output 1
- loop.

```python
def fac(0, in, out) =
    in.get() >> out.put(1) >> fac(0, in, out)
```
Implementation of *front*

*front* has two subprocesses, *read* and *write*, doing forever:

- **read** receives input $x$ from *in*.
  - If $x = 0$, output $x$ on *b*.
  - If $x > 0$, output $x$ on *b*, send $x - 1$ on *in'*.  

- **write** receives input $x$ from *b*:
  - If $x = 0$, output 1.
  - If $x > 0$, receive $y$ from *out'*, send $x \times y$ on *out*.
def front() =
  val b = Channel()
  def read() = in.get() >>= b.put(x) >>=
      if x > 0 then in'.put(x - 1) else signal >> read()

  def write() = b.get() >>=
      if x = 0 then out.put(1)
      else (out'.get() >>= y >> out.put(x * y)) >> write()

read() | write()
Program for $\text{fac}$

\[
def \text{fac}(0, \text{in}, \text{out}) = \\
\text{in}.\text{get()} \gg \text{out}.\text{put}(1) \gg \text{fac}(0, \text{in}, \text{out})
\]

\[
def \text{fac}(N, \text{in}, \text{out}) = \\
\text{val} \ (\text{in}', \text{out}') = (\text{Channel}(), \text{Channel}())
\]

\[
def \text{front}() = \cdots
\]

$\text{front}() \mid \text{fac}(N - 1, \text{in}', \text{out}')$
Combining Server Farm and Pipeline

Fac_(N−1) in’ out’

front

Fac_(N)
distr coll

Fac_(N−1) in’ out’

front

Fac_(N)
Exercise: Combining Server Farm and Pipeline

- A dataset is a list of positive numbers. The datasets are available on input channel \textit{in}. Each list length is no more than \( N \), for some given \( N \).

- Required: compute mean and variance of each dataset. Output the results (as pairs) in order on channel \textit{out}.

- First, divide the processing among about \( \sqrt{N} \) servers.

- Next, structure each server as a recursive pipeline.
Recursive Equations for Mean and Variance

- Use the equations:
  
  \[
  \text{sum}([]) = 0, \\
  \text{sum}(x : xs) = x + \text{sum}(xs)
  \]

  \[
  \text{length}([]) = 0, \\
  \text{length}(x : xs) = 1 + \text{length}(xs)
  \]

  \[
  \text{mean}(xs) = \frac{\text{sum}(xs)}{\text{length}(xs)}
  \]

  \[
  \text{var}([]) = 0, \\
  \text{var}(xs) = \text{mean}(\text{map}(\text{square}, xs)) - \text{mean}(xs)^2
  \]

- Hint: For each list, compute the sum, sum of squares, and length by a recursive pipeline.
  Apply a function to compute mean and variance from these data.
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.
Scan over array $a$; swap

- $lr(i)$ returns the smallest index $j$, $i \leq j \leq t$, where $t$ is given, such that $a(i)? > p$. Returns $t + 1$ if there is no such index.

- $rl(i)$ returns the largest index $j$, $0 \leq j \leq i$, such that $a(i)? \leq p$. There is guaranteed to be such an index.

- $swap(a, b)$ swaps the contents of two refs, and returns a signal.

```python
def lr(i) = if (i <: t && a(i)? <= p) then lr(i + 1) else i
def rl(i) = if (a(i)? :> p) then rl(i - 1) else i
def swap(a, b) = (a?, b?) > (x, y) > (a := y, b := x) >> signal
```
Partition

\[
\text{def } \text{part}(p, s, t) = \quad -- \text{ } s \text{ and } t \text{ are array boundaries}
\]
\[
\text{def } \text{lr}(i) = \quad \text{if } (i <: t \&\& a(i)? <= p) \text{ then } \text{lr}(i + 1) \text{ else } i
\]
\[
\text{def } \text{rl}(i) = \quad \text{if } (a(i)? :> p) \text{ then } \text{rl}(i - 1) \text{ else } i
\]
\[
\text{val } (s', t') = (\text{lr}(s), \text{rl}(t))
\]
\[
\begin{align*}
( & \text{If}(s' + 1 <: t') \Rightarrow \text{swap}(a(s'), a(t')) \Rightarrow \text{part}(p, s' + 1, t' - 1) \\
| & \text{If}(s' + 1 = t') \Rightarrow \text{swap}(a(s'), a(t')) \Rightarrow s' \\
| & \text{If}(s' + 1 :> t') \Rightarrow t'
\end{align*}
\]

Returns \( m \) where
\[
egin{align*}
a(s) \cdots a(m) & \leq p, \\
a(m + 1) \cdots a(t) & > p
\end{align*}
\]
Sorting

def sort(s, t) =

    if s >= t then signal

    else part(a(s)?, s + 1, t) >m>

        swap(a(m), a(s)) ≫

        (sort(s, m − 1), sort(m + 1, t)) ≫

        signal

    sort(0, a.length() − 1)
Putting the Pieces together

```python
def quicksort(a) =

def swap(a, b) = (a?, b?) >(x, y)> (a := y, b := x) » signal
def part(p, s, t) =
def lr(i) = if (i <: t & & a(i)? <= p) then lr(i + 1) else i
def rl(i) = if (a(i)? :> p) then rl(i − 1) else i
val (s', t') = (lr(s), rl(t))
( Ift(s' + 1 <: t') » swap(a(s'), a(t')) » part(p, s' + 1, t' − 1)
 | Ift(s' + 1 = t') » swap(a(s'), a(t')) » s'
 | Ift(s' + 1 :> t') » t'
)
def sort(s, t) =
if s >= t then signal
else part(a(s)?, s + 1, t) »m»
    swap(a(m), a(s)) »
    (sort(s, m − 1), sort(m + 1, t)) »
signal
sort(0, a.length() − 1)
```
Remarks and Proof outline

• Concurrency without locks

• $sort(m, n)$ sorts the segment; does not touch items outside the segment.

• Then, $sort(s, m - 1)$ and $sort(m + 1, t)$ are non-interfering.

• $part(p, s, t)$ does not modify any value outside this segment. May read values.
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 \( i \)

\( parent: \) 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.

\( dfs(i, xs): \) starts a depth-first search from all nodes in \( xs \) in order,
\( i \) has a parent (or \( i = N \)),
\( xs \subseteq conn(i), \)
All nodes in \( conn(i) - xs \) have parents already.
Depth-first search

\[\text{val } N = 6 \quad \text{ -- } \text{N is the number of nodes in the graph}\]
\[\text{val } \text{parent} = \text{Table}(N, \lambda (_\_\_) = \text{Ref}(-1))\]

\[\text{def } \text{dfs}(_, [\_]) = \text{signal}\]
\[\text{def } \text{dfs}(i, x : xs) = \]
\[\quad \text{if } (\text{parent}(x)? \geq 0) \text{ then } \text{dfs}(i, xs)\]
\[\quad \text{else } \text{parent}(x) := i \gg \text{dfs}(x, \text{conn}(x)) \gg \text{dfs}(i, xs)\]

\[\text{dfs}(N, [0]) \quad \text{ -- depth-first search from node 0}\]
Shortest path problem

• Directed graph; non-negative weights on edges.

• Find shortest path from source to sink.

We calculate just the length of the shortest path.
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 function $Succ()$

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

**Figure: Graph Structure**
Recording the values

For node $u$, record its path length in channel $u$.

$u$ is a bounded channel of length 1.

The first “put” blocks all other puts until the recorded value is read out.
Algorithm

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

Goal:

eval(source, 0) | read the value recorded for the sink
Algorithm (contd.)

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

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

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

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

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

\{- Goal :-\} \quad \text{eval}(\text{source}, 0) \mid \text{sink.get()}

- Any call to \text{eval}(u, t): Length of a path from source to \(u\) is \(t\).
- First call to \text{eval}(u, t): Length of the shortest path from source to \(u\) is \(t\).
- \text{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.