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 `xs`
Return all subsequences of `xs` that sum to `n`.

`sum(5,[1,2,1,2]) = [1,2,2], [2,1,2]`
`sum(5,[1,2,1])` is silent

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

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

$$\begin{aligned}\mathit{def}\ \ & sum(n, x : xs) = \\ & \quad sum(n - x, xs)\ \ {>}ys{>}\ x : ys \\ & \ \mid sum(n, xs)\end{aligned}$$

Backtracking: Use of Otherwise

Given: integer `n`, list of integers `xs`
Return the "first" subsequence of `xs` that sums to `n`.

`sum(5,[1,2,1,2]) = [1,2,2]`
`sum(5,[1,2,1])` is silent

$$def \;\; sum(0, \_) = [\,]$$

$$def \;\; sum(\_, [\,]) = stop$$

$$def \;\; sum(n, x : xs) = x : sum(n - x, xs) \;\; ; \;\; sum(n, xs)$$

Backtracking: Eight queens

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

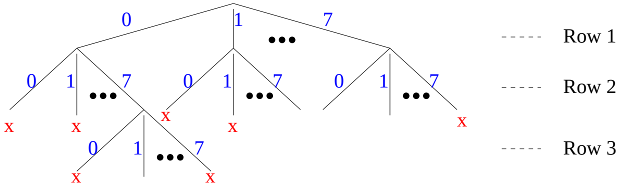


Figure: Backtrack Search for Eight queens

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 $extend([\,])$.

$$
\begin{array}{l}
def \;\; extend(xs) = \\
\quad if \;\; (length(xs) = 8) \; then \;\; xs \\
\quad else \;\; (open(xs) \;\; >j> \; extend(j : xs))
\end{array}
$$

Using Closure

A UNITY Program

$$x, y = 0, 0$$

$$\begin{aligned} & x < y \rightarrow x := x + 1 \\ | \; & y := y + 1 \end{aligned}$$

- 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

$$\textit{val}\ \ (x, y) = \ (\textit{Ref}(0), \textit{Ref}(0))$$

$$\textit{def}\ \ f1() = \ \textit{Ift}(x? <: y?) \gg x := x? + 1$$
$$\textit{def}\ \ f2() = \ y := y? + 1$$

Run the program by:

- choosing a function arbitrarily,
- choosing functions fairly.

Scheduling the UNITY Program

$def\ \ unity(fs) =$
$val\ \ arlen = \ length(fs)$
$val\ \ fnarray = \ Array(arlen)$

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

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

{- Initiate the work -}
$populate(0, fs) \gg exec()$

Running the example program

$$val \;\; (x, y) = (Ref(0), Ref(0))$$

$$def \;\; f1() = Ift(x? <: y?) \gg x := x? + 1$$
$$def \;\; f2() = y := y? + 1$$

$$unity([f1, f2])$$

Associative Fold

- Define $afold(f, xs)$ where f is an associative binary function and xs 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.

$$
\begin{aligned}
&\mathit{def}\ \ \mathit{afold}(f,[x]) = x\\
&\mathit{def}\ \ \mathit{afold}(f,xs) =\\
&\qquad \mathit{def}\ \ \mathit{step}([\,]) = [\,]\\
&\qquad \mathit{def}\ \ \mathit{step}([x]) = [x]\\
&\qquad \mathit{def}\ \ \mathit{step}(x:y:xs) = f(x,y) : \mathit{step}(xs)\\
&\qquad \mathit{afold}(f,\mathit{step}(xs))
\end{aligned}
$$

- $f(x,y) : \mathit{step}(xs)$ is an implicit fork-join.
- $f(x,y)$ executes concurrently with $\mathit{step}(xs)$.
- All calls to f execute concurrently within each iteration of afold.

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.

$$
\begin{array}{l}
\textit{def}\ \ \textit{acfold}(f, \textit{xs}) = \\
\quad \textit{val}\ \ c = \ \textit{Channel}() \\
\\
\quad \textit{def}\ \ \textit{xfer}([\,]) = \ \textit{stop} \\
\quad \textit{def}\ \ \textit{xfer}(x : \textit{xs}) = \ (c.\textit{put}(x), \textit{xfer}(\textit{xs})) \\
\\
\quad \textit{def}\ \ \textit{combine}(1) = \ c.\textit{get}() \\
\quad \textit{def}\ \ \textit{combine}(m) = \\
\qquad\quad c.\textit{get}()\ \ {>}x{>}\ c.\textit{get}()\ \ {>}y{>} \\
\qquad\quad (c.\textit{put}(f(x,y))\ \gg\ \textit{stop}\ \mid\ \textit{combine}(m-1)) \\
\\
\quad \textit{xfer}(\textit{xs})\ \mid\ \textit{combine}(\textit{length}(\textit{xs}))
\end{array}
$$

map-reduce

- 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.
  $process(t)$ processes a single task t.
  $process(t)$ 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

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{processlist}([\,]) = [\,] \\
\mathit{def}\ \ \mathit{processlist}(t : ts) = \mathit{process}(t) : \mathit{processlist}(ts) \\
\\
\mathit{def}\ \ \mathit{process}(t) = \\
\quad \mathit{val}\ \ \mathit{processor} = \mathit{Processorpool}() \\
\quad \mathit{val}\ \ (\mathit{result}, b) = (\mathit{processor}(t), \mathit{true}) \mid (\mathit{Rwait}(T), \mathit{false}) \\
\quad \mathit{if}\ b\ \mathit{then}\ \ \mathit{result}\ \mathit{else}\ \ \mathit{process}(t) \\
\\
\mathit{processlist}(\mathit{tasks})\ \ >x>\ \mathit{reduce}(x)
\end{array}
$$

Parsing using Recursive Descent

Consider the grammar:

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

Parsing strategy

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

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

To avoid multiple publications (in ambiguous grammars),

$$\begin{array}{l} \text{def}\ \ isexpr(xs) = \\ \quad \text{val}\ \ res = expr(xs)\ \ >[\,]>\ true\ \ ;\ \ false \\ \quad res \end{array}$$

- - - - - - - - - - - - - Test

$isexpr$
$(["(","(","3","*","3",")",")","+","(","3","+","3",")"])$
— ((3*3))+(3+3)

$::\ \ true$

Function for each non-terminal

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

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

Exception Handling

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

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{request}(x) = \\
\quad \mathit{val}\ \ \mathit{exc} = \ \mathit{Channel}() \ \text{-- returns a channel site} \\
\\
\qquad \mathit{server}(x, \mathit{exc}) \\
\quad \mid \mathit{exc.get}() \ >r> \mathit{exc.put}(\mathit{auth}(r)) \ \gg \ \mathit{stop}
\end{array}
$$

Process Networks

- 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 $c.put$ and $c.get$ as site calls.
- In our examples, $c.get$ is blocking and $c.put$ 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:

$$def \;\; p(c,e) = c.get() \;\; >x> \;\; Compute(x) \;\; >y> \;\; e.put(y) \;\; \gg \;\; p(c,e)$$

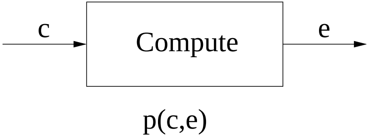


Figure: Iterative Process

Composing Processes into a Network

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

$$\mathit{def}\ \ net(c, d, e) = \ \ p(c, e) \mid p(d, e)$$

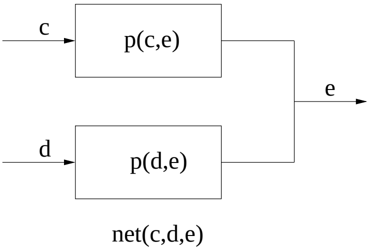


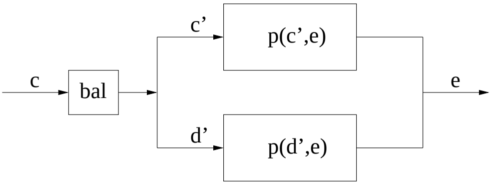
Figure: Network of Iterative Processes

Workload Balancing

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

$$
\begin{aligned}
\mathit{def}\ \ bal(c, c', d') = \quad & c.get()\ \ {>}x{>}\ random(2)\ \ {>}t{>} \\
& (\mathit{if}\ \ t = 0\ \mathit{then}\ \ c'.put(x)\ \mathit{else}\ \ d'.put(x))\ \gg \\
& bal(c, c', d')
\end{aligned}
$$

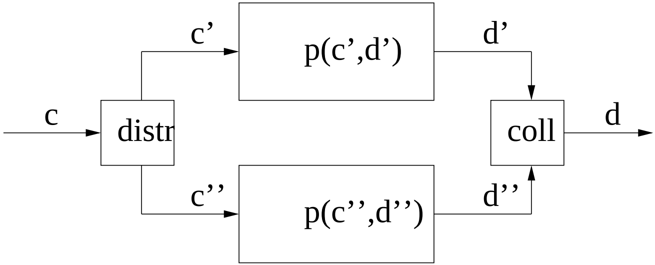
$$
\begin{aligned}
\mathit{def}\ \ workbal(c, e) = \ \ & \mathit{val}\ \ c' = \ Channel() \\
& \mathit{val}\ \ d' = \ Channel() \\
& bal(c, c', d')\ |\ net(c', d', e)
\end{aligned}
$$



workBal(c,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.

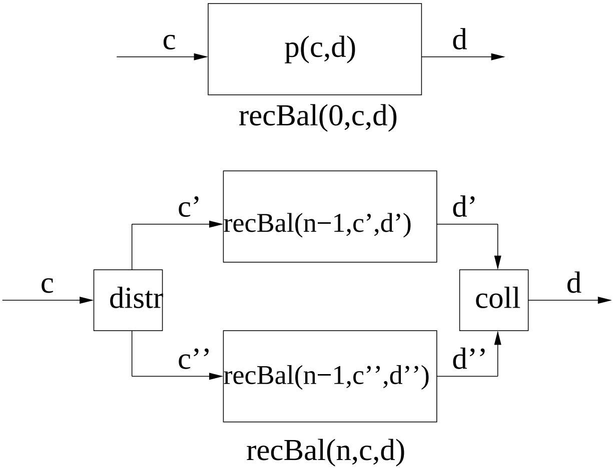


Deterministic Load Balancing

$$
\begin{array}{l}
\textit{def}\ \ detbal(in, out) = \\
\quad \textit{def}\ \ distributor(c, c', c'') = \\
\quad\quad c.get()\ \ >x>\ c'.put(x)\ \gg \\
\quad\quad c.get()\ \ >y>\ c''.put(y)\ \gg \\
\quad\quad distributor(c, c', c'') \\
\\
\quad \textit{def}\ \ collector(d', d'', d) = \\
\quad\quad d'.get()\ \ >x>\ d.put(x)\ \gg \\
\quad\quad d''.get()\ \ >y>\ d.put(y)\ \gg \\
\quad\quad collector(d', d'', d) \\
\\
\quad \textit{val}\ \ (in', in'') =\ (Channel(), Channel()) \\
\quad \textit{val}\ \ (out', out'') =\ (Channel(), Channel()) \\
\\
\quad\quad distributor(in, in', in'')\ |\ collector(out', out'', out) \\
\quad |\ p(in', out')\ |\ p(in'', out'')
\end{array}
$$

Deterministic Load Balancing with 2^n servers

Construct the network recursively.



Recursive Load Balancing Network

$$\textit{def}\ \ recbal(0, in, out) = P(in, out)$$

$$\begin{array}{l} \textit{def}\ \ recbal(n, in, out) = \\ \quad \textit{def}\ \ distributor(c, c', c'') = \cdots \\ \\ \quad \textit{def}\ \ collector(d', d'', d) = \cdots \\ \\ \quad \textit{val}\ \ (in', in'') = (Channel(), Channel()) \\ \quad \textit{val}\ \ (out', out'') = (Channel(), Channel()) \\ \\ \qquad distributor(in, in', in'') \mid collector(out', out'', out) \\ \quad \mid recbal(n-1, in', out') \mid recbal(n-1, in'', out'') \end{array}$$

An Iterative Process: Transducer

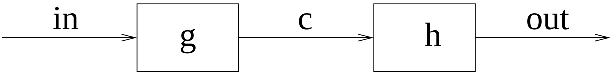
Compute $f(x)$ for each x in channel in and output to out, in order.

$$\begin{aligned} &def\ \ transducer(in, out, fn) = \\ &\quad in.get()\ \ {>}x{>}\ \ out.put(fn(x)) \gg transducer(in, out, fn) \end{aligned}$$

Pipeline network

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

$$
\begin{array}{l}
\color{red}{def}\ \ \color{blue}{pipe(in, out, g, h)}\ \color{red}{=} \\
\qquad \color{red}{val}\ \ \color{blue}{c}\ \color{red}{=}\ \ \color{blue}{Channel()} \\
\qquad \color{blue}{transducer(in, c, g)}\ \ \color{red}{|}\ \ \color{blue}{transducer(c, out, h)}
\end{array}
$$

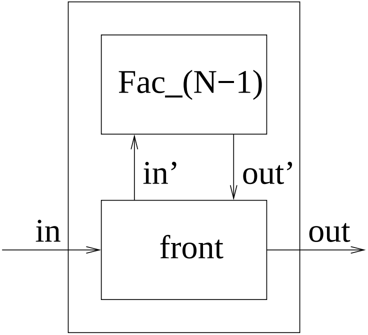


Recursive Pipeline network

Consider computing factorial of each input.

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

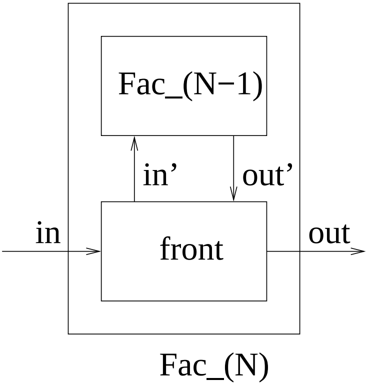
Suppose $x \leq N$, for some given N.



Fac_(N)

Outline of a program

$$
\begin{array}{l}
\textit{def}\;\; \mathit{fac}(N, \mathit{in}, \mathit{out}) = \\
\quad \textit{val}\;\; (\mathit{in}', \mathit{out}') = (\mathit{Channel}(), \mathit{Channel}()) \\
\quad \mathit{front}(\mathit{in}, \mathit{out}, \mathit{in}', \mathit{out}') \mid \mathit{fac}(N-1, \mathit{in}', \mathit{out}')
\end{array}
$$



Implementation of Fac_0

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

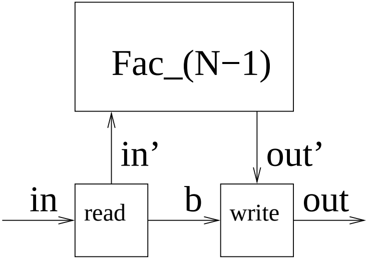
$$
\begin{aligned}
&\mathit{def}\ \ \mathit{fac}(0, \mathit{in}, \mathit{out}) = \\
&\quad \mathit{in}.\mathit{get}() \gg \mathit{out}.\mathit{put}(1) \gg \mathit{fac}(0, \mathit{in}, \mathit{out})
\end{aligned}
$$

Implementation of *front*

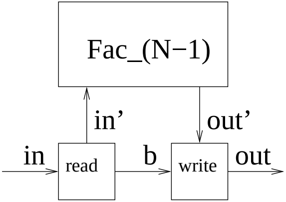
front has two subprocesses, read and write, doing forever:

- 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



Code of *front*



$$\mathit{def}\ \ \mathit{front}() =$$
$$\quad \mathit{val}\ \ b = \ \mathit{Channel}()$$
$$\quad \mathit{def}\ \ \mathit{read}() = \ \mathit{in.get}()\ \ {>}x{>}\ b.\mathit{put}(x)\ \gg$$
$$\qquad \mathit{if}\ \ x :> 0\ \mathit{then}\ \ \mathit{in}'.\mathit{put}(x-1)\ \mathit{else}\ \ \mathit{signal}\ \gg \mathit{read}()$$

$$\quad \mathit{def}\ \ \mathit{write}() = \ b.\mathit{get}()\ \ {>}x{>}$$
$$\qquad \mathit{if}\ \ x = 0\ \mathit{then}\ \ \mathit{out.put}(1)$$
$$\qquad \mathit{else}\ \ (\mathit{out}'.\mathit{get}()\ \ {>}y{>}\ \mathit{out.put}(x*y))\ \gg \mathit{write}()$$

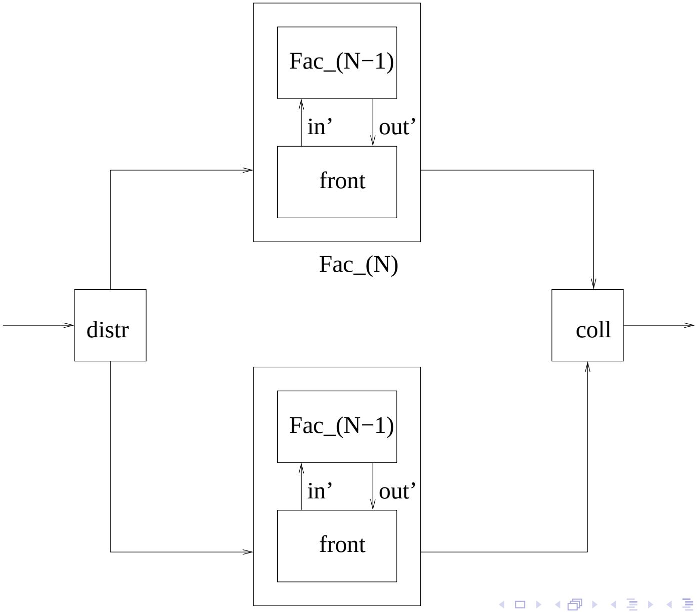
$$\mathit{read}()\ |\ \ \mathit{write}()$$

Program for fac

$$
\begin{aligned}
&\mathit{def}\ \ \mathit{fac}(0, \mathit{in}, \mathit{out}) = \\
&\quad \mathit{in}.\mathit{get}() \gg \mathit{out}.\mathit{put}(1) \gg \mathit{fac}(0, \mathit{in}, \mathit{out})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{def}\ \ \mathit{fac}(N, \mathit{in}, \mathit{out}) = \\
&\quad \mathit{val}\ \ (\mathit{in}', \mathit{out}') = (\mathit{Channel}(), \mathit{Channel}()) \\
&\\
&\quad \mathit{def}\ \ \mathit{front}() = \cdots \\
&\\
&\mathit{front}() \mid \mathit{fac}(N-1, \mathit{in}', \mathit{out}')
\end{aligned}
$$

Combining Server Farm and Pipeline



Exercise: Combining Server Farm and Pipeline

- A dataset is a list of positive numbers.
  The datasets are available on input channel 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 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:

$$
\begin{aligned}
&\mathit{sum}([\,]) = 0,\\
&\mathit{sum}(x : xs) = x + \mathit{sum}(xs)\\[1em]
&\mathit{length}([\,]) = 0,\\
&\mathit{length}(x : xs) = 1 + \mathit{length}(xs)\\[1em]
&\mathit{mean}(xs) = \mathit{sum}(xs)/\mathit{length}(xs)\\[1em]
&\mathit{var}([\,]) = 0,\\
&\mathit{var}(xs) = \mathit{mean}(\mathit{map}(\mathit{square}, xs)) - \mathit{mean}(xs) \ast\ast 2
\end{aligned}
$$

- 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 guranteed to be such an index.
- $swap(a,b)$ swaps the contents of two refs, and returns a signal.

$$def\;\; lr(i) = \;\; if\;\; (i <: t\ \&\&\ \ a(i)? <= p)\ then\;\; lr(i+1)\ else\;\; i$$

$$def\;\; rl(i) = \;\; if\;\; (a(i)? :> p)\ then\;\; rl(i-1)\ else\;\; i$$

$$def\;\; swap(a,b) = \;\; (a?, b?)\;\; >(x,y)>\;\; (a := y,\;\; b := x)\;\; \gg\;\; signal$$

Partition

$def\ \ part(p, s, t) =$ -- s and t are array boundaries

$\quad def\ \ lr(i) = \ \ if\ (i <: t\ \&\&\ \ a(i)? <= p)\ then\ \ lr(i+1)\ else\ \ i$

$\quad def\ \ rl(i) = \ \ if\ (a(i)? :> p)\ then\ \ rl(i-1)\ else\ \ i$

$\quad val\ \ (s', t') = (lr(s), rl(t))$

$\quad (\ \ Ift(s'+1 <: t') \gg swap(a(s'), a(t')) \gg part(p, s'+1, t'-1)$

$\quad \mid Ift(s'+1 = t') \ \ \gg swap(a(s'), a(t')) \gg s'$

$\quad \mid Ift(s'+1 :> t') \gg t'$

$\quad)$

Returns m where

$$a(s) \cdots a(m) \leq p,$$
$$a(m+1) \cdots a(t) > p$$

Sorting

$def\ \ sort(s, t) =$

$\quad if\ s >= t\ then\ \ signal$

$\quad else\ part(a(s)?, s + 1, t)\ \ {>}m{>}$

$\qquad\quad swap(a(m), a(s))\ \ \gg$

$\qquad\quad (sort(s, m - 1), sort(m + 1, t))\ \ \gg$

$\qquad\quad signal$

$sort(0, a.length() - 1)$

Putting the Pieces together

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{quicksort}(a) = \\
\quad \mathit{def}\ \ \mathit{swap}(a,b) = \ (a?, b?)\ \ {>}(x,y){>}\ (a := y,\ b := x)\ \gg\ \mathit{signal} \\
\quad \mathit{def}\ \ \mathit{part}(p,s,t) = \\
\qquad \mathit{def}\ \ \mathit{lr}(i) = \ \ \mathit{if}\ (i <: t\ \&\&\ \ a(i)? <= p)\ \mathit{then}\ \ \mathit{lr}(i+1)\ \mathit{else}\ \ i \\
\qquad \mathit{def}\ \ \mathit{rl}(i) = \ \ \mathit{if}\ (a(i)? :> p)\ \mathit{then}\ \ \mathit{rl}(i-1)\ \mathit{else}\ \ i \\
\qquad \mathit{val}\ \ (s', t') = \ (\mathit{lr}(s), \mathit{rl}(t)) \\
\qquad (\ \ \mathit{Ift}(s'+1 <: t')\ \gg\ \mathit{swap}(a(s'), a(t'))\ \gg\ \mathit{part}(p, s'+1, t'-1) \\
\qquad \mid \mathit{Ift}(s'+1 = t')\ \ \gg\ \mathit{swap}(a(s'), a(t'))\ \gg\ s' \\
\qquad \mid \mathit{Ift}(s'+1 :> t')\ \gg\ t' \\
\qquad) \\
\quad \mathit{def}\ \ \mathit{sort}(s,t) = \\
\qquad \mathit{if}\ s >= t\ \mathit{then}\ \ \mathit{signal} \\
\qquad \mathit{else}\ \mathit{part}(a(s)?, s+1, t)\ \ {>}m{>} \\
\qquad\qquad\quad \mathit{swap}(a(m), a(s))\ \gg \\
\qquad\qquad\quad (\mathit{sort}(s, m-1), \mathit{sort}(m+1, t))\ \gg \\
\qquad\qquad\quad \mathit{signal} \\
\mathit{sort}(0, a.\mathit{length}() - 1)
\end{array}
$$

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
$parent(i) = v$, $v >= 0$, 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

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

$\textit{def}\ \ \textit{dfs}(\_, [\,]) = \ \textit{signal}$

$\textit{def}\ \ \textit{dfs}(i, x : \textit{xs}) =$
$\quad \textit{if}\ \ (\textit{parent}(x)? >= 0)\ \textit{then}\ \ \textit{dfs}(i, \textit{xs})$
$\quad \textit{else}\ \ \textit{parent}(x) := i \gg \textit{dfs}(x, \textit{conn}(x)) \gg \textit{dfs}(i, \textit{xs})$

$\textit{dfs}(N, [0])$ - - 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 $Succ()$

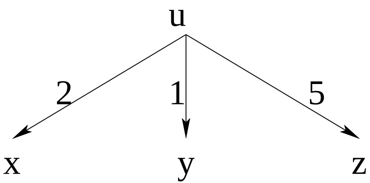


Figure: Graph Structure

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

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

$Goal$: $eval(source, 0) \mid$
read the value recorded for the *sink*

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*

---

def $eval(u, t) =$ $u.put(t) \gg$
$Succ(u) \; >(v, d)>$
$Rwait(d) \gg$
$eval(v, t + d)$

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

Algorithm(contd.)

$$\begin{aligned} \mathit{def}\ \ \mathit{eval}(u,t) = \quad & u.\mathit{put}(t) \gg \\ & \mathit{Succ}(u)\ >(v,d)> \\ & \mathit{Rwait}(d) \gg \\ & \mathit{eval}(v, t+d) \end{aligned}$$

$$\texttt{\{-}\ \ \mathit{Goal}: \texttt{-\}} \qquad \mathit{eval}(\mathit{source}, 0) \mid \mathit{sink}.\mathit{get}()$$

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