Structured Concurrent Programming

William Cook
Jayadev Misra
David Kitchin
John Thywissen
Arthur Peters

Department of Computer Science
University of Texas at Austin

http://orc.csres.utexas.edu

The 10th International Symposium on Formal Aspects of Component Software
Jiangxi Normal University, Nanchang, China.
October 28 - 30, 2013

Structured Concurrent Programming

- Structured Sequential Programming: Dijkstra circa 1968
  Component Integration in a sequential world.

- Structured Concurrent Programming:
  Component Integration in a concurrent world.

Traditional approaches to handling Concurrency

- Adding concurrency to serial languages:
  - Threads with mutual exclusion using semaphore.
  - Transaction.
- Process Networks.

Orc

- Orc addresses Design: as a component integration system.

  Components:

  - from many vendors
  - for many platforms
  - written in many languages
  - may run concurrently and in real-time

Evolution of Orc

- Web-service Integration
- Component Integration
- Structured Concurrent Programming

Web-service Integration: Internet Scripting

- Contact two airlines simultaneously for price quotes.
- Buy a ticket if the quote is at most $300.
- Buy the cheapest ticket if both quotes are above $300.
- Buy a ticket if the other airline does not give a timely quote.
- Notify client if neither airline provides a timely quote.

-

Enhanced Goal: Component Integration

Components could be:

- Web services
- Library modules
- Custom Applications, including real time

Components could be for:

- Functional Transformation
- Data Object Creation
- Real-time Computation

Component Integration; contd.

- Combine any kind of component, not just web services
- Small components: add two numbers, print a file ...
- Large components: Linux, MSword, email server, file server ...
- Time-based components: for real-time computation
- Actuators, sensors, humans as components
- Fast and Slow components
- Short-lived and Long-lived components
- Written in any language for any platform

Concurrency

- Component integration: typically sequential using objects
- Concurrency is ubiquitous
- Magnitude higher in complexity than sequential programming
- No generally accepted method to tame complexity
- May affect security

Orc: Structured Concurrent Programming

- A combinator combines two components to get a component
- Combinators may be applied recursively
- Results in hierarchical/modular program construction
- Combinators may orchestrate components concurrently
- Orc is just about 4 combinators

Power of Orc

- Solve all known synchronization, communication problems
- Code objects, active objects
- Solve all known forms of real-time and periodic computaions
- Solve a limited kind of transactions
- and, all combinations of the above

Some Typical Applications

- Adaptive Workflow (Business process management):
  Workflow lasting over months or years
  Security, Failure, Long-lived Data

- Extended 911:
  Using humans as components
  Components join and leave
  Real-time response

- Network simulation:
  Experiments with differing traffic and failure modes
  Animation

Some Typical Applications, contd.

- Grid Computations
- Music Composition
- Traffic simulation
- Computation Animation
- Robotics

Some Typical Applications, contd.

- Map-Reduce using a server farm
- Thread management in an operating system
- Mashups (Internet Scripting).
- Concurrent Programming on Android.

Some Very Large Applications

- Logistics
- Managing Olympic Games
- Smart City

Current Status

- Strong Theoretical Basis
- An elegant programming language
  - as good as functional on functional problems
  - can work with mutable store, real-time dependent components, non-determinacy
  - concurrency
  - hierarchical, modular, recursive
- Robust Implementation
  - Run program through a Web browser or locally
  - Web site: `orc.csres.utexas.edu`
  - Several papers, Ph.D. thesis
- Several Chapters of a book

Concurrent orchestration in Haskell

John Launchbury and Trevor Elliott
Proceedings of the third ACM Haskell symposium on Haskell

Orc Calculus

- Site: Basic service or component.
- Concurrency combinators for integrating sites.
- Calculus includes nothing other than the combinators.

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

New concepts are programmed using new sites.

Examples of Sites

- $+ \quad - \quad * \quad \&\& \quad || \quad =$ ...
- `Println`, `Random`, `Prompt`, `Email` ...
- Mutable Ref, Semaphore, Channel, ...
- `Timer`
- External Services: Google Search, MySpace, CNN, ...
- Any Java Class instance, Any Orc Program
- Factory sites; Sites that create sites: `Semaphore`, `Channel` ...
- Humans

  ...

Sites

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

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
for some x from g do f	$f <x< g$	Pruning
if f halts without publishing do g	$f \; ; \; g$	Otherwise

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
for some x from g do f	$f <x< g$	Pruning
if f halts without publishing do g	$f\ ;\ g$	Otherwise

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
for some x from g do f	$f <x< g$	Pruning
if f halts without publishing do g	$f\ ;\ g$	Otherwise

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
for some x from g do f	$f <x< g$	Pruning
if f halts without publishing do g	$f\ ;\ g$	Otherwise

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
for some x from g do f	$f < x < g$	Pruning
if f halts without publishing do g	$f \; ; \; g$	Otherwise

Symmetric composition: $f \mid g$

- Evaluate f and g independently.
- Publish all values from both.
- No direct communication or interaction between f and g. They can communicate only through sites.

  Example: $CNN(d) \mid BBC(d)$

  Calls both CNN and BBC simultaneously.
  Publishes values returned by both sites. (0, 1 or 2 values)

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 $f \ >x> \ (g \ >y> \ h)$

Schematic of Sequential composition

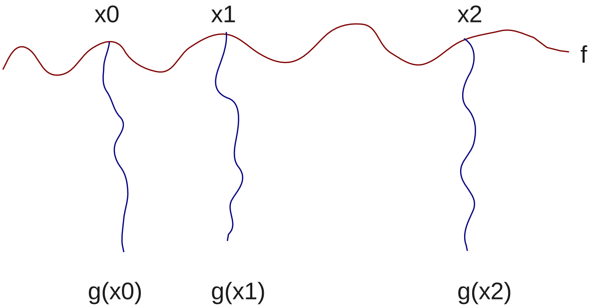


Figure: Schematic of f $>x>$ g

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 $(f \;<x<\; g) \;<y<\; h$

Example of Pruning

$$Email(address, x) \;\; <x< \; (CNN(d) \mid BBC(d))$$

Binds x to the first value from $CNN(d) \mid BBC(d)$.
Sends at most one email.

Multiple Pruning happens concurrently

$add(x,y) \; <x< f \; <y< g$ is $(add(x,y) \; <x< f \;) \; <y< g$

$(add(x,y) \; <x< f \;)$ is computed concurrently with g

$(add(x,y)$, f and g computed concurrently.

Otherwise: f ; g

Do f. If f halts without publishing then do g.

- An expression halts if
  - its execution can take no more steps, and
  - all called sites have either responded, or will never respond.
- A site call may respond with a value, indicate that it will never respond (helpful), or do neither.
- All library sites in Orc are helpful.

Examples of $f \; ; \; g$

- $1 \; ; \; 2$ publishes 1

- $(CNN(d) \mid BBC(d)) \; >x> \; Email(address, x) \; ; \; Retry()$

  If the sites are never helpful, this is equivalent to

  $(CNN(d) \mid BBC(d)) \; >x> \; Email(address, x)$

- $5/0$; "Exception leads to Halt" publishes

  "Exception leads to Halt"

Orc program

- Orc program has
  - a goal expression,
  - a set of definitions.
- The goal expression is executed. Its execution
  - calls sites,
  - publishes values.

Some Fundamental Sites

- $Ift(b)$, $Iff(b)$: boolean b,
  Returns a signal if b is true/false; remains silent otherwise.
  Site is helpful: indicates when it will never respond.

- $Rwait(t)$: integer t, $t \geq 0$, returns a signal t time units later.

- $stop$: never responds. Same as $Ift(false)$ or $Iff(true)$.

- $signal$: returns a signal immediately.
  Same as $Ift(true)$ or $Iff(false)$.

Use of Fundamental Sites

- Print all publications of h. When h halts, publish "done".

  $$h \; >x> \; Println(x) \gg stop \; ; \; \text{"}done\text{"}$$

- Timeout:
  Call site M.
  Publish its response if it arrives within 10 time units.
  Otherwise publish 0.

  $$x \; <x< \; (M() \mid Rwait(10) \gg 0)$$

Interrupt f

- Evaluation of f can not be directly interrupted.
- Introduce two sites:
  - $Interrupt.set$: to interrupt f
  - $Interrupt.get$: responds only after $Interrupt.set$ has been called.

  - $Interrupt.set$ is similar to $release$ on a semaphore;
    $Interrupt.get$ is similar to $acquire$ on a semaphore.
- Instead of f, evaluate

$$z \; {<}z{<} \; (f \mid Interrupt.get())$$

Site Definition

$$\mathit{def}\ \ \mathit{MailOnce}(a) = \\ \quad \mathit{Email}(a, m)\ \ {<}m{<}\ (\mathit{CNN}(d) \mid \mathit{BBC}(d))$$

$$\mathit{def}\ \ \mathit{MailLoop}(a, t) = \\ \quad \mathit{MailOnce}(a)\ \gg\ \mathit{Rwait}(t)\ \gg\ \mathit{MailLoop}(a, t)$$

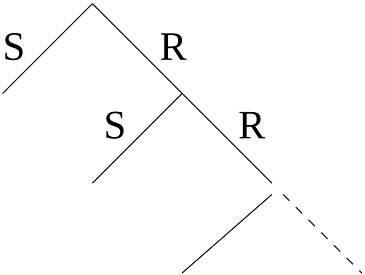
$$\mathit{def}\ \ \mathit{metronome}() =\ \mathit{signal} \mid (\mathit{Rwait}(1) \gg \mathit{metronome}())$$

- Expression is called like a procedure.
  It may publish many values. *MailLoop* does not publish.

Example of a Definition: Metronome

Publish a signal every unit.

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



Unending string of Random digits

$metronome() \gg Random(10)$ – one every unit

$def\ rand\_seq(dd) =$ – at a specified rate
$Random(10) \mid Rwait(dd) \gg rand\_seq(dd)$

Example of Site call

- Site $Query()$ returns a value (different ones at different times).
- Site $Accept(x)$ returns x if x is an acceptable value; it is silent otherwise.
- Call $Query$ every second forever and publish all its acceptable values.

$$metronome() \gg Query() \; >x> \; Accept(x)$$

Concurrent Site call

- Sites are often called concurrently.
- Each call starts a new instance of site execution.
- If a site accesses shared data, concurrent invocations may interfere.

Example: Publish each of "tick" and "tock" once per second, "tock" after an initial half-second delay.

$$
\begin{array}{lll}
 & metronome() \gg \text{"}tick\text{"} \\
\mid Rwait(500) \gg & metronome() \gg \text{"}tock\text{"}
\end{array}
$$

Logical Connectives; 2-valued Logic

And: Publish a signal if both sites do.
Or: Publish a signal if either site does.

$M() \gg N()$ – “and”

$b\ {<}b{<}\ (M()\ |\ N())$ – “or”

$M()\ ;\ N()$ – “or” with helpful M

$(M() \gg \mathit{true}\ ;\ \mathit{false})\ {>}b{>}\ \mathit{Iff}(b)$ – “not” with helpful M

Parallel or

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

$$val\ \ x = f$$
$$val\ \ y = g$$

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

Parallel or; contd.

Compute the parallel or and return just one value:

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

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

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

Airline quotes: Application of Parallel or

- Contact airlines A and B.
- Return any quote if it is below \$300 as soon as it is available, otherwise return the minimum quote.
- $threshold(x)$ returns x if $x < 300$; silent otherwise.
  $Min(x, y)$ returns the minimum of x and y.

  val $z =$
      val $x = A()$
      val $y = B()$

      $threshold(x) \mid threshold(y) \mid Min(x, y)$
  z

Choice: Execute either f or g

$$\textit{if}\ (\textit{true}\ \mid\ \textit{false})\ \textit{then}\ f\ \textit{else}\ g$$

Simple definitions using $\mathit{Random}()$

- Return a random boolean.

  $\mathit{def}\ \ \mathit{rbool}() = (\mathit{Random}(2) = 0)$

- Return a random real number between 0 and 1.

  $\mathit{def}\ \ \mathit{frandom}() = \mathit{Random}(1001)/1000.0$

- Return true with probability p, false with $(1 - p)$

  $\mathit{def}\ \ \mathit{biasedBool}(p) = (\mathit{Random}(1000) <: p * 1000)$

Timeout

Publish M's response if it arrives before time t,
Otherwise, publish 0.

$$z \; {<}z{<} \; (M() \mid (Rwait(t) \gg 0)), \text{ or}$$

$$val \;\; z = M() \mid (Rwait(t) \gg 0)$$
$$z$$

Fork-join parallelism

Call sites M and N in parallel.
Return their values as a tuple after both respond.

$$((u, v) \quad <u< M()) \quad <v< N()$$

or,

$$(M(), N())$$

Simple Parallel Auction

- A list of bidders in a sealed-bid, single-round auction.
- $b.ask()$ requests a bid from bidder b.
- Ask for bids from all bidders, then publish the highest bid.

$$\begin{aligned}&def\ \ auction([\,]) = 0\\&def\ \ auction(b : bs) = max(b.ask(), auction(bs))\end{aligned}$$

Notes:

- All bidders are called simultaneously.
- If some bidder fails, then the auction will never complete.

Parallel Auction with Timeout

- Take a bid to be 0 if no response is received from the bidder within 8 seconds.

$$
\begin{array}{l}
\textit{def}\ \ auction([\,]) = \ 0 \\
\\
\textit{def}\ \ auction(b : bs) = \\
\quad max(\\
\qquad\qquad b.ask() \mid (Rwait(8000) \gg 0), \\
\qquad\qquad auction(bs) \\
\qquad\quad)
\end{array}
$$

Identities of $|$, $\gg$, $\ll$ and $;$

(Zero and $|$) $\quad f \mid stop = f$

(Commutativity of $|$) $\quad f \mid g = g \mid f$

(Associativity of $|$) $\quad (f \mid g) \mid h = f \mid (g \mid h)$

(Left zero of $\gg$) $\quad stop \gg f = stop$

(Associativity of $\gg$) if h is x-free

$$(f >x> g) >y> h = f >x> (g >y> h)$$

(Right zero of $\ll$) $\quad f \ll stop = f$

(generalization of right zero)

$$f \ll g = f \ll (stop \ll g) = f \mid (stop \ll g)$$

(relation between $\ll$ and $<x<$)

$$f \ll g = f <x< g, \quad \text{if } x \notin free(f).$$

(commutativity) $\quad (f <x< g) <y< h = (f <y< h) <x< g$

if $x \notin free(h)$, $y \notin free(g)$, and x, y are distinct.

(associativity of $;$) $\quad (f ; g) ; h = f ; (g ; h)$

Distributivity Identities

($|$ over $>x>$; left distributivity)

$$(f \mid g) \;>x>\; h = f \;>x>\; h \mid g \;>x>\; h$$

($|$ over $<x<$) $\quad (f \mid g) \;<x<\; h = (f \;<x<\; h) \mid g$, if $x \notin free(g)$.

($>y>$ over $<x<$) $\quad (f \;>y>\; g) \;<x<\; h = (f \;<x<\; h) \;>y>\; g$
if $x \notin free(g)$, and x and y are distinct.

($<x<$ over otherwise) $(f \;<x<\; g) \,;\, h = (f \,;\, h) \;<x<\; g$, if $x \notin free(h)$.

Identities that don't hold

(Idempotence of $|$) $\quad f \mid f = f$

(Right zero of $\gg$) $\quad f \gg \mathit{stop} = \mathit{stop}$

(Left Distributivity of $\gg$ over $|$)
$f \gg (g \mid h) = (f \gg g) \mid (f \gg h)$

Orc Language

- 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
- Orc combinators everywhere
- Class for active objects

Data types

- Number: $5, \quad -1, \; 2.71828, \quad -2.71e-5$
- Boolean: *true*, *false*
- String: `"orc"`, `"ceci n'est pas une |"`

$1+2$	evaluates to	3
$0.4 = 2.0/5$	evaluates to	*true*
$3-5 :> 5-3$	evaluates to	*false*
true && (*false* || *true*)	evaluates to	*true*
$3/0$	is	silent
`"Try" + "Orc"`	evaluates to	`"TryOrc"`

Variable Binding; Silent expression

$val\ \ x = \ 1 + 2$

$val\ \ y = \ x + x$

$val\ \ z = \ x/0$ -- expression is silent

$val\ \ u = \ $ if $(0 <: 5)$ then 0 else z

Exceptions

$3/0$ halts.

Conditional Expression

if	*true*	then	"blue"	else	"green"	—	is "blue"
if	"fish"	then	"yes"	else	"no"	—	is silent
if	*false*	then	4+5	else	4+true	—	is silent
if	*true*	then	0/5	else	5/0	—	is 0

Tuples

$(1 + 2, 7)$	is	$(3, 7)$
("true" + "false", $\mathit{true} \,\|\|\, \mathit{false}, \mathit{true} \,\&\&\, \mathit{false}$)	is	("truefalse", true, false)
(2/2, 2/1, 2/0)	is	silent

Lists

$[1, 2+3]$	is	$[1, 5]$
$[\mathit{true}\ \&\&\ \mathit{true}]$	is	$[\mathit{true}]$
$[\,]$	is	the empty list
$[5, 5+\mathit{true}, 5]$	is	silent

List Constructor is a colon :

$3{:}[5, 7] = [3, 5, 7]$

$3{:}[\,] = [3]$

Translating Programs to Orc Calculus

- All programs are translated to Orc calculus.
- $1 + 2$ becomes $add(1, 2)$
  All arithmetic and logical operators, tuples, lists are site calls.
  if-then-else is translated with calls to Ift, Iff sites.
- $1 + (2 + 3)$ should become $add(1, add(2, 3))$
  But this is not legal Orc! Site calls can not be nested.
- What is the meaning of $(1 \mid 2) + (2 \mid 3)$?

Orc Combinators everywhere

Parameters in site calls could be Orc expressions

$$(1 + 2) \mid (2 + 3)$$

$$(1 \mid 2) + (2 \mid 3)$$

Implicit Concurrency

- An experiment tosses two dice.
  Experiment is a success if and only if sum of the two dice thrown is 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

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{toss}() = \ \mathit{add}(x,1) \ \ <x< \mathit{Random}(6) \\
\mathit{def}\ \ \mathit{exp}(n) = \\
\quad (\ \mathit{Ift}(b) \ \gg 0 \\
\quad \mid \mathit{Iff}(b) \ \gg \\
\qquad (\ \mathit{add}(x,y) \\
\qquad\quad <x< (\ \mathit{exp}(m) \ \ <m< \mathit{sub}(n,1)\ ) \\
\qquad\quad <y< (\ \mathit{Ift}(bb) \ \gg 1 \mid \mathit{Iff}(bb) \ \gg 0\ ) \\
\qquad\qquad <bb< \mathit{equals}(p,7) \\
\qquad\qquad\quad <p< \mathit{add}(q,r) \\
\qquad\qquad\qquad <q< \mathit{toss}() \\
\qquad\qquad\qquad <r< \mathit{toss}() \\
\qquad) \\
\quad) \ \ <b< \mathit{equals}(n,0)
\end{array}
$$

Note: $2n$ parallel calls to $\mathit{toss}()$.

Deflation

- Given expression $C(\ldots, e, ..)$, single value expected at e
- translate to $C(\ldots, x, ..)$ $<x<$ e where x is fresh
- $val\ \ z = g$
  f becomes
  $f\ \ <z<\ g$
- applicable hierarchically.

$(1|2) * (10|100)$ is
$(Times(x, y)\ \ <x<\ (1 \mid 2))\ \ <y<\ (10 \mid 100)$, or
$Times(x, y)\ \ <x<\ (1 \mid 2)\ \ <y<\ (10 \mid 100)$
Implication:
Arguments of site calls are evaluated in parallel.
Note: A strict site is called when all arguments have been evaluated.

Barrier Synchronization in $M() \gg f \mid N() \gg g$

- Require: f and g start only after both M and N complete.

- Rendezvous of CSP or CCS;
  M and N are complementary actions.

$$(M(), N()) \gg (f \mid g)$$

Priority

- Publish N's response asap, but no earlier than 1 unit from now.
  Apply fork-join between $Rwait(1)$ and N.

  $$val\ \ (u, \_) = (N(), Rwait(1))$$

- Call M, N together.
  If M responds within one unit, publish its response.
  Else, publish the first response.

  $$val\ \ x = M() \mid u$$

Pattern Matching in val

(x,y) = (2+3,2*3)	binds	x to 5 and y to 6
[a,b] = ["one", "two"]	binds	a to "one", b to "two"
((a,b),c) = ((1, true), [2, false])	binds	a to 1, b to true, and c to [2, false]
(x,_,_) = (1,(2,2),[3,3,3])	binds	x to 1
[[_,x],[_,y]] = [[1,3],[2,4]]	binds	x to 3 and y to 4

Pattern Matching in Site Definition parameters

A site adds two pairs componentwise;
publishes the resulting pair.

$$\mathit{def}\ \ \mathit{pairsum}(a,b) =$$
$$a\ >(x,y)>\ b\ >(x',y')>\ (x+x',y+y')$$

or, even better,

$$\mathit{def}\ \ \mathit{pairsum}((x,y),(x',y')) =\ (x+x',y+y')$$

Pattern Matching, clausal definition

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

$$\textit{def}\;\; sum(x : xs) = x + sum(xs)$$

Clauses are evaluated in order from top to bottom.

Tree Reconstruction

1. Given a non-empty sequence of natural numbers.
2. Does the sequence represent the depths of terminal nodes in a binary tree, from left to right? Then it is valid.

Example: $[1, 3, 3, 2]$ is valid, $[1, 3, 2, 2]$ is not.

Output the tree structutre if the sequence is valid;
Output $\mathit{NonTree}()$ otherwise.

Theorem

- $[0]$ is valid.
- $[l] \mathbin{+\!\!+} x \mathbin{+\!\!+} x \mathbin{+\!\!+} [r]$, where $[l] \mathbin{+\!\!+} x$ has no duplicates, is valid iff

  $[l] \mathbin{+\!\!+} (x - 1) \mathbin{+\!\!+} [r]$ is valid.

Tree Reconstruction; Contd.

$$type\ Tree = Node(Tree, Tree) \mid Leaf() \mid NonTree()$$

$$def\ \ tc(\_, [\,]) = \ NonTree()$$

$$def\ \ tc([\,], [(v, t)]) = \ if\ (v = 0)\ then\ \ t\ else\ \ NonTree()$$

$$def\ \ tc([\,], v : right) = \ tc([v], right)$$

$$\begin{aligned} def\ \ tc((u, t) : left, (v, t') : right) &= \\ if\ u = v\ then\ \ tc(left, (v-1, Node(t, t')) : right) \\ else\ \ tc((v, t') : (u, t) : left, right) \end{aligned}$$

Typical test: $tc([\,], [(3, Leaf()), (3, Leaf()), (2, Leaf()), (2, Leaf())])$

Tree Reconstruction; contd.

Simplify input preparation:

$tc([\,], [(3, Leaf()), (3, Leaf()), (2, Leaf()), (2, Leaf())])$ replaced by

$checktree([3, 3, 2, 2])$

$$
\begin{array}{lll}
def & mklist([\,]) = & [\,] \\
def & mklist(x : xs) = & (x, Leaf()) : mklist(xs) \\
def & checktree(xs) = & tc([\,], mklist(xs))
\end{array}
$$

$checktree([3, 3, 2, 2])$

– $NonTree()$

$checktree([1, 3, 3, 2])$

– $Node(Leaf(), Node(Node(Leaf(), Leaf()), Leaf()))$

$checktree([3, 3, 2, 2, 2])$

– $Node(Node(Node(Leaf(), Leaf()), Leaf()), Node(Leaf(), Leaf()))$

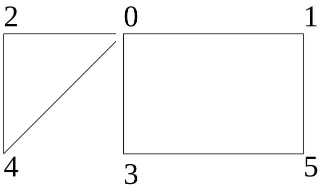
Example: Fibonacci numbers

$$\mathit{def}\ \ H(0) = (1,1)$$
$$\mathit{def}\ \ H(n) = H(n-1)\ \ >(x,y)>\ \ (y, x+y)$$

$$\mathit{def}\ \ \mathit{Fib}(n) = H(n)\ \ >(x,\_)>\ \ x$$

{- Goal expression -}
$\mathit{Fib}(5)$

Clausal Definition, Pattern Matching
Example: Defining graph connectivity



An Undirected Graph

$$
\begin{aligned}
&\mathit{def}\ \ \mathit{conn}(0) = [1, 2, 3, 4] \\
&\mathit{def}\ \ \mathit{conn}(1) = [0, 5] \\
&\mathit{def}\ \ \mathit{conn}(2) = [0, 4] \\
&\mathit{def}\ \ \mathit{conn}(3) = [0, 5] \\
&\mathit{def}\ \ \mathit{conn}(4) = [0, 2] \\
&\mathit{def}\ \ \mathit{conn}(5) = [1, 3]
\end{aligned}
$$

$\|$

$$
\begin{aligned}
&\mathit{def}\ \ \mathit{conn}(i) = \\
&\quad\ \ i > 0 > [1, 2, 3, 4] \\
&\quad \mid i > 1 > [0, 5] \\
&\quad \mid i > 2 > [0, 4] \\
&\quad \mid i > 3 > [0, 5] \\
&\quad \mid i > 4 > [0, 2] \\
&\quad \mid i > 5 > [1, 3]
\end{aligned}
$$

Sites

- Sites are first-class values.
  A site may be a parameter in site call.
  A site may return a site as a value.

  $M() \;>(x, y)>\; x(y)$ -- x, y are sites

- Sites may have methods.

  $Channel() \;>ch>\; ch.put(3)$

- Translation of method call $ch.put(3)$:

  $ch(\text{“}put'') \;>x>\; x(3)$

Closure: Sites as values

$val\ \ minmax = \ (min, max)$

========================

$def\ \ apply2((f, g), (x, y)) = \ (f(x, y), g(x, y))$

$apply2(minmax, (2, 1))$ publishes (1, 2)

========================

$def\ \ pmap(f, []) = \ []$

$def\ \ pmap(f, x : xs) = \ f(x) : pmap(f, xs)$

$pmap(lambda(i) = i * i, [2, 3, 5])$ publishes [4, 9, 25]

========================

$def\ \ repeat(f) = \ f() \gg \ repeat(f)$

$def\ \ pr() = \ Println(3)$

$repeat(pr)$ prints 3 forever.

val, tuple, closure

$$def\ \ circle() =$$

$$val\ \ pi = \ 3.1416$$

$$def\ \ perim(r) = \ 2 * pi * r$$

$$def\ \ area(r) = \ pi * r **2\ \ \#$$

$$(perim, area)$$

Some Factory Sites

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

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

$Cell()\ >r>\ (r.write(5) \mid r.read())$, or $Cell()\ >r>\ r := 5 \mid r?$

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

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

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

Simple Swap

Convention:

$$\begin{array}{ll} a? & \text{is } a.read() \\ b := x & \text{is } b.write(x) \end{array}$$

Take two references as arguments,
Exchange their values, and return a signal.

$$def\ \ swap(i,j) = (i?, j?)\ >(x,y)>\ (i := y,\ j := x)\ \gg signal$$

Note: a and b could be identical Refs.

Update linked list

Given is a one-way linked list.
Its first item is called first.
Now add value v as the first item.

$$Ref() \;\; >r> \\ r := (v, first) \;\; \gg \\ first := r$$

or,

$$Ref((v, first)) \;\; >r> \\ first := r$$

Binary Search Tree; using Ref()

$def\;\; search(key) =$ return true or false

$\quad searchstart(key) \;\; >(\_,\_,q)> \; (q \neq null)$

$def\;\; insert(key) =$ true if value was inserted, false if it was there

$\quad searchstart(key) \;\; >(p,d,q)>$

$\quad if\; q = null$

$\quad\quad then\; Ref() \;\; >r>$

$\quad\quad\quad r := (key, null, null) \cdots$

$\quad\quad else \;\; \cdots$

Array Permutation

- Randomly permute the elements of an array in place.
- $randomize(i)$ permutes the first i elements of arry a and publishes a signal.

$$
\begin{aligned}
&def\ \ permute(a) = \\
&\quad def\ \ randomize(0) = \ signal \\
&\quad def\ \ randomize(i) = \ Random(i)\ \ >j> \\
&\qquad\qquad\qquad\qquad\quad swap(a(i-1), a(j))\ \ \gg \\
&\qquad\qquad\qquad\qquad\quad randomize(i-1) \\
\\
&\quad randomize(a.length())
\end{aligned}
$$

Example: Return Array of 0-valued Semaphores

$$
\begin{aligned}
&\mathit{def}\ \ \mathit{semArray}(n) = \\
&\quad \mathit{val}\ \ a = \mathit{Array}(n) \\
&\quad \mathit{def}\ \ \mathit{populate}(0) = \mathit{signal} \\
&\quad \mathit{def}\ \ \mathit{populate}(i) = a(i-1) := \mathit{Semaphore}(0) \gg \mathit{populate}(i-1) \\
&\\
&\quad \mathit{populate}(n) \gg a
\end{aligned}
$$

Usage: $\mathit{semArray}(5)\ >a>\ a(1)?.\mathit{release}()$

Library site: *Table*

- $Table(n, f)$, where $n > 0$ and f a site closure.
  Creates site g, where $g(i) = f(i)$, $0 \leq i < n$.
  An array of site values pre-computed and reused.

- All values of g are computed at instantiation.

- Allows creating arrays of structures.

- Site f may be supplied as: $lambda(i) = h(i)$

Examples:

- $val\ \ g = \ Table(5, lambda(\_) \ = Channel() \ )$
- $val\ \ h = \ Table(5, lambda(i) \ = 2 * i)$
- $val\ \ s = \ Table(5, lambda(\_) \ = Semaphore(0) \ )$

Definition Mechanism: Class

- Encapsulate data and objects with methods
- Create new sites; Extend behaviors of existing sites
- Allow concurrent method invocation on objects (monitors)
- Create active objects with time-based behavior

Classes can be translated to Orc calculus using a special site.

Object Creation: Stack

- Define stack with methods push and pop.
- Parameter n gives the maximum stack size.
- Store the stack elements in array *store*, current stack length in *len*.
- push on a full stack or pop from an empty stack halts with no effect.

Stack definition

$$
\begin{aligned}
&def\ \ class\ Stack(n) =\\
&\quad val\ \ store = \ Table(n, lambda(\_) = Ref())\\
&\quad val\ \ len = \ Ref(0)\\
&\\
&\quad def\ \ push(x) =\\
&\qquad Ift(len? <: n) \gg store(len?) := x \gg len := len? + 1\\
&\\
&\quad def\ \ pop() =\\
&\qquad Ift(len? :> 0) \gg len := len? - 1 \gg store(len?)?
\end{aligned}
$$

`{- class Goal -}` *stop*

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

$val\ \ st = \ Stack(5)$

$st.push(3) \gg st.push(5) \gg st.pop() \gg st.pop()$

Special case: only one class instance

$$val \;\; (push, pop) = \; Stack(5) \;\; >r> \;\; (r.push, r.pop)$$

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

$$push(3) \gg push(5) \gg pop() \gg pop()$$

Class Syntax

- Class definition
  - Like site definition
  - May include parameters
- Clausal definitions allowed.
- All definitions within a class are exported.
  Such definitions are accessed as dot methods.

Class Semantics: Class is a site with methods

- A class call creates and publishes a site.
- All the rules for site definition apply except:
  - Publications of class goal expression are ignored,
  - Each method (site) publishes at most once,
  - Class calls are strict (site calls are non-strict),
  - Class method calls are not terminated prematurely by prune (follows the rule for sites).
- Methods may be invoked concurrently, as in sites.

Special attention to concurrent invocation

$$st.push(3) \gg st.pop() \gg Rwait(1000) \gg st.pop()$$
$$\mid st.push(4) \gg stop$$

- If method executions were atomic there would be some output.
- This program sometimes produces no output.
  Method executions may overlap and interfere.

Example: Matrix (with upper and lower indices)

$$\mathit{def}\ \ \mathit{class}\ \mathit{Matrix}((\mathit{row}, \mathit{row}'), (\mathit{col}, \mathit{col}')) =$$

$$\mathit{val}\ \ \mathit{mat} = \ \mathit{Array}((\mathit{row}' - \mathit{row} + 1) * (\mathit{col}' - \mathit{col} + 1))$$

$$\mathit{def}\ \ \mathit{access}(i, j) = \ \mathit{mat}((i - \mathit{row}) * (\mathit{col}' - \mathit{col} + 1) + j)$$

stop

\- - - - - - - - - - - - - - - - - - Test

$$\mathit{val}\ \ A = \ \mathit{Matrix}((-2, 0), (-1, 3)).\mathit{access}$$

$$A(-1, 2) := 5 \gg A(-1, 2) := 3 \gg A(-1, 2)?$$

A Matrix of Classes

$$def\ \ class\ CMatrix((row, row'), (col, col'), cap) =$$

$$val\ \ mat = \ Table((row' - row + 1) * (col' - col + 1), cap)$$

$$def\ \ access(i, j) = \ mat((i - row) * (col' - col + 1) + j)$$

$$stop$$

- - - - - - - - - - - - - - - - - - Test; A matrix of Channels

$$val\ \ A = \ CMatrix((-2, 0), (-1, 3), lambda(\_) = Channel()).access$$

$$A(-1, 2).put(3) \ \gg\ A(-1, 2).get()$$

Create a new site: Cell using Semaphore and Ref

$def \;\; class\ Cell() =$

$val \;\; s = \;\; Semaphore(1)$
$val \;\; r = \;\; Ref()$

$def \;\; write(v) = \;\; s.acquire() \gg r := v$

$def \;\; read() = \;\; r?$ -- $r?$ blocks until r has been written

$stop$

New Site: Bounded Channel

- Bounded channel of size n may block for *put* and *get*.
- Use semaphore $p =$ number of empty positions.
- Use *Channel* to hold data items.

Bounded Channel; contd.

$def\ \ class\ BChannel(n) =$
$\quad val\ \ b = \ Channel()$
$\quad val\ \ p = \ Semaphore(n)$

$\quad def\ \ put(x) = \ p.acquire() \ \gg \ b.put(x)$

$\quad def\ \ get() = \ b.get() \ \ {>}x{>}\ p.release() \ \gg \ x$

$\quad stop$

Extend functionality of a site: add length method to Channel

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{class}\ \mathit{Channel}'() = \\
\quad \mathit{val}\ \ \mathit{ch} = \ \mathit{Channel}() \\
\quad \mathit{val}\ \ \mathit{chlen} = \ \mathit{Counter}(0) \\
\\
\quad \mathit{def}\ \ \mathit{put}(x) = \ \mathit{ch.put}(x) \gg \mathit{chlen.inc}() \\
\quad \mathit{def}\ \ \mathit{get}() = \ \mathit{ch.get}() \ > x > \mathit{chlen.dec}() \gg x \\
\quad \mathit{def}\ \ \mathit{len}() = \ \mathit{chlen.value}() \\
\\
\quad \mathit{stop}
\end{array}
$$

\- - - - - - - - - - - - - - - - - - Test

$$\mathit{val}\ \ c = \ \mathit{Channel}'()$$

$$
\begin{array}{l}
c.\mathit{put}(1000) \gg c.\mathit{put}(2000) \gg \mathit{Println}(c.\mathit{len}()) \gg \\
c.\mathit{get}() \gg \mathit{Println}(c.\mathit{len}()) \gg \mathit{stop}
\end{array}
$$

Memoization

For site f (with no arguments) cache its value after the first call.

res: stores the cached value.
s: semaphore value is 0 if the site value has been cached.

$$
\begin{array}{l}
val\ \ res = \ Cell() \\
val\ \ s = \ Semaphore(1) \\
def\ \ memo() = \\
\qquad val\ \ z = \ res? \mid \ s.acquire() \gg res := f() \gg stop \\
\qquad z
\end{array}
$$

Note: Concurrent calls handled correctly.

Memoize an argument site using Class

$def\ \ class\ Memo(f) =$

$\quad val\ \ res = \ Cell()$

$\quad val\ \ s = \ Semaphore(1)$

$\quad def\ \ memo() =$

$\quad\quad val\ \ z = \ res? \mid \ s.acquire() \gg res := f() \gg stop$

$\quad\quad\quad z$

$\quad stop$

— Usage

$val\ \ prandom = \ Memo(lambda() = Random(20)).memo$

$prandom() \mid prandom() \mid prandom()$

Concurrent access: Client-Server interaction

- Asynchronous protocol for client-server interaction.
- At most one client interacts at a time with the server.
- Client requests service and supplies input data.
- Server reads data, computes and writes out the result.
- Client receives result.

Client-Server interaction API

- $req(x)$:
  Performed by the client to send data to the server.
  Client receives a response when the operation completes.
  The operation may remain blocked forever.

- $read()$:
  For the server to remove the data sent by the client.
  The operation is blocked if there is no outstanding request.

- $write(v)$:
  Server returns v as the response to the client.
  Operation is non-blocking.

Client-Server interaction; Program

$def \;\; class\ csi() =$

$val \;\; sem = \; Semaphore(1)$
$val \;\; (u, v) = \; (Channel(), Channel())$
-- sem ensures that only one client interacts at a time
-- client data stored in u, server response in v

$def \;\; req(x) = \; sem.acquire() \; \gg$
$\quad u.put(x) \; \gg \; v.get() \; >y> $
$\quad sem.release() \; \gg \; y$

$def \;\; read() = \; u.get()$

$def \;\; write(x) = \; v.put(x)$

$stop$

Examples

- Combinatorial
- Mutable store manipulation
- Synchronization, Communication

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

List map

$$def\ \ parmap(\_, [\,]) = \ [\,]$$

$$def\ \ parmap(f, x : xs) = \ f(x) : parmap(f, xs)$$

List map (Contd.)

$$def\ \ seqmap(\_, [\,]) = \ [\,]$$

$$def\ \ seqmap(f, x : xs) = \ f(x)\ \ {>}y{>}\ (y : seqmap(f, xs))$$

Infinite Set Enumeration

Enumerate all finite binary strings.
A binary string is a list of 0,1.

$$
\begin{array}{ll}
\mathit{def}\ \ \mathit{bin}() = & \\
 & [\,] \\
| & \mathit{bin}()\ \ {>}\mathit{xs}{>}\ (0 : \mathit{xs} \mid 1 : \mathit{xs})
\end{array}
$$

Note: Unguarded recursion.

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

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

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

$$\begin{aligned} def \;\; & parsum(n, x : xs) = \\ & \quad parsum(n - x, xs) \;\; {>}ys{>} \; x : ys \\ & \;\; | \;\; parsum(n, xs) \end{aligned}$$

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

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

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

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

Subset Sum (Contd.), Concurrent Backtracking

Publish the first sublist of xs that sums to n.

Run the searches concurrently.

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

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

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

Note: Neither search in the last clause may succeed.

Mutual Recursion: Finite state transducer

Convert an input string:

- Remove all white spaces in the beginning.
- Reduce all other blocks of white spaces (consecutive white spaces) to a single white space.

```
---Mary---had-a--little--lamb-
```

becomes (where `-` denotes a white space)

```
Mary-had-a-little-lamb-
```

A finite State Transducer

A deterministic Finite State Machine. No concurrency.

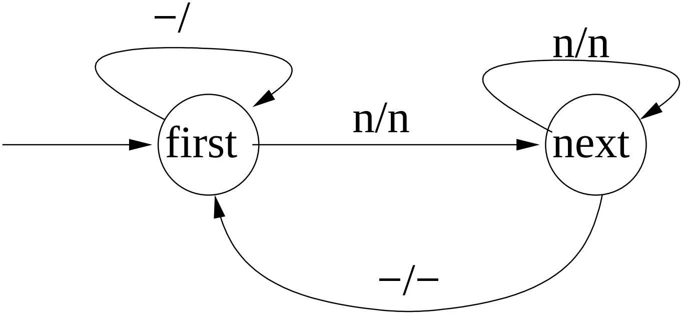


Figure: n is a symbol other than white space

A Program

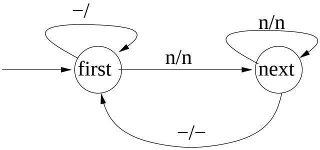


Figure: n is a symbol other than white space

$$
\begin{aligned}
&def\ \ first([\,]) = [\,]\\
&def\ \ first(\text{" "} : xs) = first(xs)\\
&def\ \ first(x : xs) = x : next(xs)
\end{aligned}
$$

$$
\begin{aligned}
&def\ \ next([\,]) = [\,]\\
&def\ \ next(\text{" "} : xs) = \text{" "} : first(xs)\\
&def\ \ next(x : xs) = x : next(xs)
\end{aligned}
$$

Non-deterministic search: String Matching

- Given a pattern string p and a text string t, determine if p occurs in t (as a contiguous substring).
- Run two searches simultaneously:
  Is p a prefix of t?
  Is p in the string excluding the first symbol of t?
- Terminate the search if either is a success.

Helper Sites

- *parallelOr*: to terminate the search asap.
- $prefix(xs, ys)$ returns true if and only if xs is a prefix of ys.
  (strings are given as lists of symbols).

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{parallelOr}(y, z) = \\
\quad \mathit{val}\ \ r = \ \mathit{Ift}(y) \gg \mathit{true} \ \mid \ \mathit{Ift}(z) \gg \mathit{true} \ \mid \ y \,||\, z \\
\quad r
\end{array}
$$

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{prefix}([\,], ys) = \ \mathit{true} \\
\mathit{def}\ \ \mathit{prefix}(xs, [\,]) = \ \mathit{false} \\
\mathit{def}\ \ \mathit{prefix}(x : xs, y : ys) = \ (x = y)\ \&\&\ \mathit{prefix}(xs, ys)
\end{array}
$$

String Matching Program

- $stringmatch(xs, ys)$ returns true if and only if xs is a contiguous substring of ys.
  (strings are given as lists of symbols).

$$def \;\; stringmatch([\,], ys) = true$$

$$def \;\; stringmatch(xs, [\,]) = false$$

$$
\begin{aligned}
&def \;\; stringmatch(xs, y : ys) = \\
&\quad parallelOr \\
&\qquad (stringmatch(xs, ys), \\
&\qquad\; prefix(xs, y : ys) \\
&\qquad)
\end{aligned}
$$

Using Closure

A UNITY Program

$$x, y = 0, 0$$

$$x < y \rightarrow x := x + 1$$
$$\mid y := y + 1$$

- 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

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

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

Run the program by:

- choosing a function arbitrarily,
- choosing functions fairly.

Scheduling the UNITY Program

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

{- $populate()$ 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

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

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

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

Fold on a non-empty list

fold with binary f: $fold(+, [x_0, x_1, \cdots]) = x_0 + x_1 \cdots$

$$def \;\; fold(_, [x]) = \; x$$

$$def \;\; fold(f, x : xs) = \; f(x, fold(xs))$$

Associative fold on a non-empty list

$$
\begin{aligned}
&def\;\; afold(f,[x]) = x\\
&def\;\; afold(f,xs) =
\end{aligned}
$$

$$
\begin{aligned}
&def\;\; pairfold([\,]) = [\,]\\
&def\;\; pairfold([x]) = [x]\\
&def\;\; pairfold(x : y : xs) = f(x,y) : pairfold(xs)
\end{aligned}
$$

$$afold(f, pairfold(xs))$$

map and associative fold: *map_afold*

Associative commutative fold over a channel

A channel has two methods: put and get.

$chFold(c, n), n > 0$, folds the first n items of channel c and publishes.

$$def \;\; chFold(c, 1) = c.get()$$

$$def \;\; chFold(c, n) = f(chFold(c, n/2), chFold(c, n - n/2))$$

Does not combine values computed in different halves, even when they are available quickly.

Associative commutative fold over a channel; contd.

$$def\ \ comb(0) = \ stop$$

$$def\ \ comb(1) = f(c.get(), c.get())\ \ {>}x{>}\ c.put(x)\ \gg stop$$

$$def\ \ comb(k) = comb(1) \mid comb(k-1)$$

$$comb(n-1)$$

- $comb(k)$ combines $k+1$ values from the channel and puts the result back in the channel. Does not publish.
- If number of items, n, in the channel is strictly more than k, $comb(k)$ terminates.
- So, $comb(n-1)$ combines n values from the channel and puts the result back in the channel, and halts.

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.

- Site *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 site *reduce*.

map-reduce

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

Parsing using Recursive Descent

Consider the grammar:

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

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*

Site for each non-terminal

Given: $expr ::= term \mid term + expr$

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

$$\text{def } expr(xs) = term(xs) > ys > (ys \mid ys > \text{"+"} : zs > expr(zs))$$

$$\text{def } term(xs) = factor(xs) > ys > (ys \mid ys > \text{"*"} : zs > term(zs))$$

$$\text{def } factor(xs) = literal(xs) \mid xs > \text{"("} : ys > expr(ys) > \text{")"} : zs > zs$$

$$\text{def } literal(n : xs) = n > \text{"3"} > xs \mid n > \text{"5"} > xs$$

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

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.
- A segment is given by a pair of indices (u, v). Elements in the segment are: $a(u)..a(v-1)$. Segment length is $v - u$ if $v \geq u$.
- $segmentsort(u, v)$ sorts a segment 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.

$$def\ \ swap(i, j) = (i?, j?) \ >(x, y)> (i := y,\ j := x) \gg signal$$

$$def\ \ quicksort(a) =$$
$$\quad def\ \ segmentsort(u, v) =$$
$$\quad\quad if\ v - u > 1\ then$$
$$\quad\quad\quad part(a(u)?, u, v)\ \ >m>$$
$$\quad\quad\quad swap(a(u), a(m))\ \gg$$
$$\quad\quad\quad (segmentsort(u, m), segmentsort(m + 1, v)) \gg signal$$
$$\quad\quad else\ \ signal$$
$$segmentsort(0, a.length?)$$

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?)
```

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

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

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

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

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

Sequential Breadth-First Traversal of a Graph

N nodes in a graph,

$root$ a specified node,

$succ(x)$ is the list of successors of x,

Publish the $parent$ of each node in Breadth-First Traversal.

$$
\begin{array}{l}
def\ \ bfs(N, root, succ) = \\
\qquad val\ \ parent = Table(N, lambda(\_) = Cell()) \\
\\
\qquad -\ bfs' \text{ is } bfs \text{ on a list of nodes} \\
\qquad def\ \ bfs'([\,]) = signal \\
\qquad def\ \ bfs'(x : xs) = bfs'(append(xs, expand(x))) \\
\\
parent(root) := N \gg bfs'([root]) \gg parent
\end{array}
$$

Site *expand*

$def\ \ expand(x) =$

– $expand'(x, ys)$, ys successors of x yet to be scanned

$$def\ \ expand'(\_, [\,]) = [\,]$$
$$def\ \ expand'(x, z : zs) =$$
$$(parent(z) := x \ \gg \ z : expand'(x, zs)) \ ; \ expand'(x, zs)$$

$expand'(x, succ(x))$

Sequential Breadth-First Traversal: Complete Program

$$
\begin{array}{l}
\textit{def}\ \ \textit{bfs}(N, \textit{root}, \textit{succ}) = \\
\quad \textit{val}\ \ \textit{parent} = \ \textit{Table}(N, \textit{lambda}(\_) = \textit{Cell}()) \\
\\
\quad \textit{def}\ \ \textit{expand}(x) = \\
\qquad \textit{def}\ \ \textit{expand}'(\_, [\,]) = \ [\,] \\
\qquad \textit{def}\ \ \textit{expand}'(x, z : zs) = \\
\qquad\quad (\textit{parent}(z) := x \gg z : \textit{expand}'(x, zs))\ \ ;\ \ \textit{expand}'(x, zs) \\
\quad \textit{expand}'(x, \textit{succ}(x)) \qquad\qquad \text{– Goal of expand} \\
\\
\quad \textit{def}\ \ \textit{bfs}'([\,]) = \ \textit{signal} \\
\quad \textit{def}\ \ \textit{bfs}'(x : xs) = \ \textit{bfs}'(\textit{append}(xs, \textit{expand}(x))) \\
\\
\textit{parent}(\textit{root}) := N \gg \textit{bfs}'([\textit{root}]) \gg \textit{parent}
\end{array}
$$

Concurrent Breadth-First Traversal

```
def bfs(N, root, succ) =
    val parent = Table(N, lambda(_) = Cell())

    def expand(x) =
      if succ(x) = [] then []
      else map_afold
        (
          lambda(y) = parent(y) := x ≫ [y] ; [],
          append,
          succ(x)
        )

    def bfs'([]) = signal
    def bfs'(xs) = bfs'(map_afold(expand, append, xs))

parent(root) := N ≫ bfs'([root]) ≫ parent
```

Memoization

Memoize calls to $f()$.

$$\mathit{val}\ \ \mathit{done} = \mathit{Cell}()$$
$$\mathit{val}\ \ \mathit{res} = \mathit{Cell}()$$

$$\mathit{def}\ \ \mathit{memof}() =$$
$$\quad \mathit{res}? \ll (\mathit{done} := \mathit{signal} \gg \mathit{res} := f())$$

Memoization of Fibonacci

$$
\begin{aligned}
&\mathit{val}\ \ N = 100\\
&\mathit{val}\ \ \mathit{done} = \mathit{Table}(N+1, \mathit{lambda}(\_) = \mathit{Cell}())\\
&\mathit{val}\ \ \mathit{res} = \mathit{Table}(N+1, \mathit{lambda}(\_) = \mathit{Cell}())
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{def}\ \ \mathit{mfib}(0) = 0\\
&\mathit{def}\ \ \mathit{mfib}(1) = 1\\
&\mathit{def}\ \ \mathit{mfib}(i) =\\
&\qquad \mathit{res}(i)?\ \ll\\
&\qquad (\mathit{done}(i) := \mathit{signal} \gg \mathit{res}(i) := \mathit{mfib}(i-1) + \mathit{mfib}(i-2))
\end{aligned}
$$

Note: Concurrent calls to $\mathit{mfib}(i)$, for each i.

Exception Handling

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

$$\mathit{def}\ \ \mathit{request}(x) =$$
$$\quad \mathit{val}\ \ \mathit{exc} = \ \mathit{Channel}()$$ -- returns a channel site

$$\quad\quad \mathit{server}(x, \mathit{exc})$$
$$\quad \mid \mathit{exc}.\mathit{get}()\ \ {>}r{>}\ \mathit{exc}.\mathit{put}(\mathit{auth}(r))\ \gg\ \mathit{stop}$$

Synchronization, Communication

`Semaphore(n)`	Semaphore with initial value `n`
`BoundedChannel(n)`	bounded (asynchronous) channel of size `n`
`Counter()`	Methods `inc()`, `dec()` and `onZero()`

$$Semaphore(1)\ \ >s>\ s.acquire()\ \gg\ r := 5\ \gg\ s.release()$$

$$BoundedChannel(1)\ \ >ch>\ (ch.put(5)\ |\ ch.put(3))$$

$$Counter()\ \ >ctr>\ (ctr.inc()\ \gg ctr.onZero()\ |\ Rwait(10)\ \gg ctr.dec())$$

Rendezvous

$$
\begin{array}{l}
def\ \ class\ zeroChannel() = \\
\quad val\ \ s = \ Semaphore(0) \\
\quad val\ \ w = \ BoundedChannel(1) \\
\quad def\ \ put(x) = \ s.acquire() \gg w.put(x) \\
\quad def\ \ get() = \ s.release() \gg w.get() \\
stop
\end{array}
$$

Pure Rendezvous

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{class}\ \mathit{pairSync}() = \\
\quad \mathit{val}\ \ s = \ \mathit{Semaphore}(0) \\
\quad \mathit{val}\ \ t = \ \mathit{Semaphore}(0) \\
\\
\quad \mathit{def}\ \ \mathit{put}() = \ s.\mathit{acquire}() \ \gg \ t.\mathit{release}() \\
\quad \mathit{def}\ \ \mathit{get}() = \ s.\mathit{release}() \ \gg \ t.\mathit{acquire}() \\
\mathit{stop}
\end{array}
$$

n-party Rendezvous

- n parties participate in a rendezvous.
- Each party (optionally) contributes some data.
- After all parties have contributed:
  a given function is applied to transform input list to output list,
  then i receives the i^{th} item of output list, and proceeds.
- Access Protocol:
  i calls $go(i, x)$ with i and data x.
  Receives its result as the response of the call.

Examples of Data Transformations

- $n = 2$: first input data item becomes the second output item.
  The classical sender-receiver paradigm.

- $n = 2$: input data items are swapped.
  Data exchange;
  can simulate the classical sender-receiver.

- Arbitrary n: every output item is the first input data item.
  Broadcast paradigm.

- Arbitrary n: secret sharing.

- Arbitrary n: i^{th} output is the rank of the i^{th} input.

Implementation Strategy

- Tables in and out hold the inputs and outputs. Each table entry is $BoundedChannel(1)$.
- $go(i, x)$ stores x in $in(i)$ if it is empty.
  Then waits to receive result from $out(i)$.
- $manager$ receives all n inputs, applies the given function and stores the results in out.

n-party Rendezvous Program

$$
\begin{aligned}
&\mathit{def}\ \ \mathit{class\ Rendezvous}(n, f) = \\
&\quad \mathit{val}\ \ \mathit{in} = \ \mathit{Table}(n, \mathit{lambda}(\_) = \mathit{BoundedChannel}(1)\ ) \\
&\quad \mathit{val}\ \ \mathit{out} = \ \mathit{Table}(n, \mathit{lambda}(\_) = \mathit{BoundedChannel}(1)\ ) \\
\\
&\quad \mathit{def}\ \ \mathit{go}(i, x) = \ \mathit{in}(i).\mathit{put}(x)\ \gg \mathit{out}(i).\mathit{get}() \\
\\
&\quad \mathit{def}\ \ \mathit{collect}(0) = \ [\,] \\
&\quad \mathit{def}\ \ \mathit{collect}(i) = \ \mathit{in}(n-i).\mathit{get}()\ :\ \mathit{collect}(i-1) \\
\\
&\quad \mathit{def}\ \ \mathit{distribute}(\_, 0) = \ \mathit{signal} \\
&\quad \mathit{def}\ \ \mathit{distribute}(v : \mathit{vl}, i) = \ \mathit{out}(n-i).\mathit{put}(v)\ \gg \mathit{distribute}(\mathit{vl}, i-1) \\
\\
&\quad \mathit{def}\ \ \mathit{manager}() = \\
&\qquad\quad \mathit{collect}([\,], n)\ \ {>}\mathit{vl}{>}\ \mathit{distribute}(f(\mathit{vl}), n)\ \gg\ \mathit{manager}() \\
\\
&\quad \mathit{manager}()
\end{aligned}
$$

Test

$$def \;\; rotate([a, b, c]) = [b, c, a]$$

$$val \;\; rg3 = Rendezvous(3, rotate).go$$

$$
\begin{array}{ll}
\;\; rg3(0,0) & >x> (\text{"0 gets "} + x) \\
|\; rg3(1,1) & >x> (\text{"1 gets "} + x) \\
|\; rg3(2,4) & >x> (\text{"2 gets "} + x) \\
|\; rg3(2,2) & >x> (\text{"2 gets "} + x)
\end{array}
$$

- - - - - - - - - - - Output

"0 gets 1"
"1 gets 4"
"2 gets 0"

Phase Synchronization

- A set of threads execute a sequence of phases.
- Required: a thread may start a phase only if all threads have finished the previous phase.
- A thread calls $nextphase()$ after each phase, and waits to receive a *signal* to execute its next phase.

Typical Usage:

$$def\ \ class\ phaseSync(n) = \cdots$$
$$val\ \ barrier = phaseSync(3).nextphase$$

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

$$Println(0.1) \gg barrier() \gg Println(0.2) \gg barrier() \gg Println(0.3)$$
$$|\ Println(1.1) \gg barrier() \gg Println(1.2) \gg barrier() \gg stop$$
$$|\ Println(2.1) \gg barrier() \gg stop$$

Implementation Strategy

- Employ two semaphores: *insem*, *outsem*, initial values 0.
- Each call to *nextphase*() increments *insem* and attempts to acquire *outsem*.
- A manager attempts to acquire *insem* n times, then releases *outsem* n times, then repeats these steps.

Program: Phase Synchronization

```
def class phaseSync(n) =
  val (insem, outsem) = (Semaphore(0), Semaphore(0))

  def nextphase() = insem.release() ≫ outsem.acquire()

  def repeat(_, 0) = signal
  def repeat(f, i) = f() ≫ repeat(i − 1, f)

  def manager() =
      repeat(insem.acquire, n) ≫
      repeat(outsem.release, n) ≫
      manager()

  manager()
```

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, from readers and writers.

Readers-Writers API

- Readers call $start(true)$, Writers $start(false)$ 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 = *ctr.value*, where *ctr* is a counter.
- On each iteration, *manager* picks the next queue entry.
  If a reader: grant access and increment *ctr*.
  If a writer:
    wait until all readers complete (*ctr*'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

$$val\ \ req = \ Channel()$$
$$val\ \ na = \ Counter()$$

$$def\ \ startread() =$$
$$\quad val\ \ s = \ Semaphore(0)$$
$$\quad req.put((true, s))\ \gg\ s.acquire()$$

$$def\ \ startwrite() =$$
$$\quad val\ \ s = \ Semaphore(0)$$
$$\quad req.put((false, s))\ \gg\ s.acquire()$$

$$def\ \ endread() = \ na.dec()$$

$$def\ \ endwrite() = \ na.dec()$$

Reader-Writer; Main Loop

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

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

$$def \;\; grant((false, s)) = \quad \text{– Writer}$$
$$na.onZero() \gg na.inc() \gg s.release() \gg na.onZero()$$

Note on Callback

- Let request queue entry be (b, f), where f is a site.
- Manager executes $f()$ for callback.
- For Readers-Writers, f is $s.release()$

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

- The queue now holds a sequence of booleans,
  true for each reader, false for each writer.

- Dispense with the queue.

- Introduce a class that has *put*, *get* methods.
  It internally maintains Ref variables, *nr* and *nw*.
  nr is the number of readers, *nw* writers.

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

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:

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

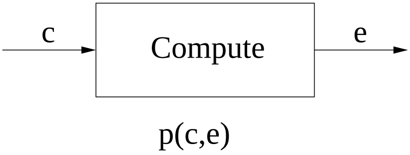


Figure: Iterative Process

Composing Processes into a Network

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

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

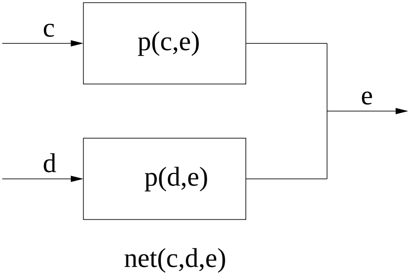


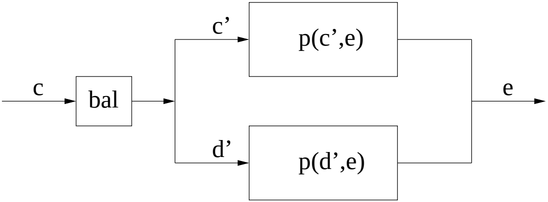
Figure: Network of Iterative Processes

Workload Balancing

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

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

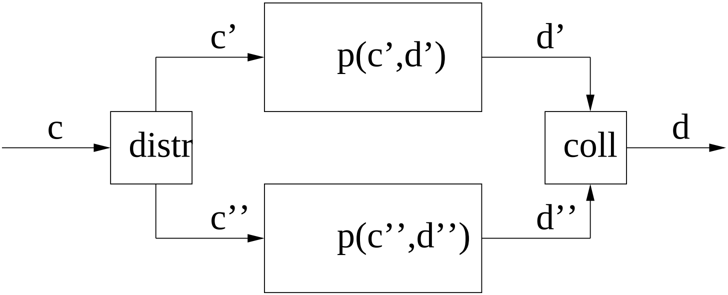
$$\begin{aligned} def\ \ workbal(c, e) = \quad & val\ \ c' = \ Channel() \\ & val\ \ d' = \ Channel() \\ & bal(c, c', d') \mid net(c', d', e) \end{aligned}$$



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.

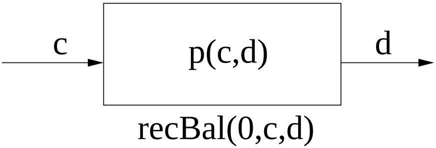


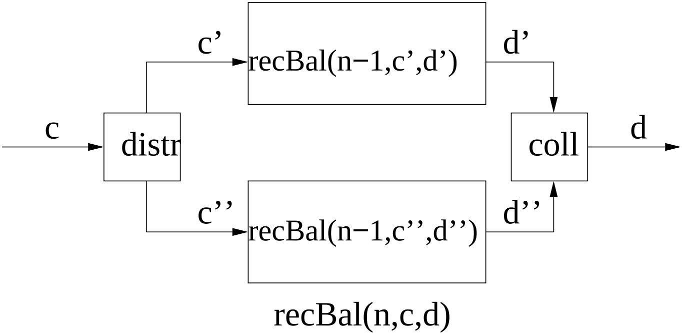
Deterministic Load Balancing

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

Deterministic Load Balancing with 2^n servers

Construct the network recursively.





Recursive Load Balancing Network

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

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

$$\quad \textit{def}\ \ distributor(c, c', c'') = \cdots$$

$$\quad \textit{def}\ \ collector(d', d'', d) = \cdots$$

$$\quad \textit{val}\ \ (in', in'') = (Channel(), Channel())$$
$$\quad \textit{val}\ \ (out', out'') = (Channel(), Channel())$$

$$\quad\quad distributor(in, in', in'') \mid collector(out', out'', out)$$
$$\quad \mid recbal(n-1, in', out') \mid recbal(n-1, in'', out'')$$

An Iterative Process: Transducer

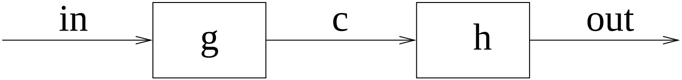
Compute $f(x)$ for each x in channel in and output to out, in order.

$$\mathit{def}\ \ \mathit{transducer}(in, out, fn) =$$
$$in.get()\ \ {>}x{>}\ \ out.put(fn(x))\ \gg\ \mathit{transducer}(in, out, fn)$$

Pipeline network

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

$$
\begin{aligned}
&\mathit{def}\ \ \mathit{pipe}(\mathit{in}, \mathit{out}, g, h) = \\
&\quad \mathit{val}\ \ c = \ \mathit{Channel}() \\
&\quad \mathit{transducer}(\mathit{in}, c, g)\ \ |\ \ \mathit{transducer}(c, \mathit{out}, h)
\end{aligned}
$$

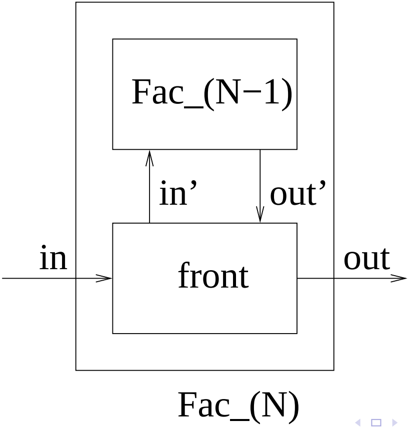


Recursive Pipeline network

Consider computing factorial of each input.

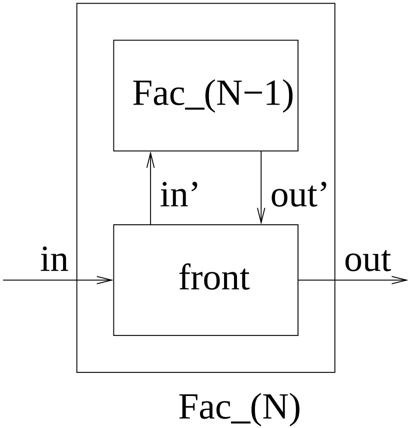
$$fac(x) = \begin{cases} 1 & \text{if} \quad x = 0 \\ x \times fac(x-1) & \text{if} \quad x > 0 \end{cases}$$

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



Outline of a program

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



Implementation of Fac_0

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

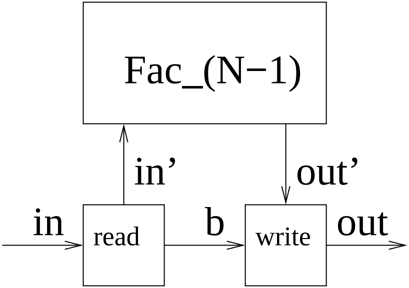
$$\mathit{def}\ \ \mathit{fac}(0, \mathit{in}, \mathit{out}) = $$
$$\mathit{in}.\mathit{get}() \gg \mathit{out}.\mathit{put}(1) \gg \mathit{fac}(0, \mathit{in}, \mathit{out})$$

Implementation of *front*

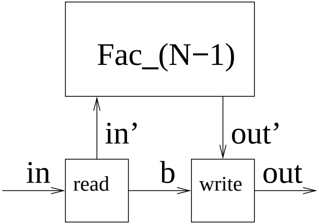
front has two subprocesses, read and write, doing forever:

- 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



Code of *front*



$def \;\; front() =$

$\quad val \;\; b = \; Channel()$

$\quad def \;\; read() = \; in.get() \;\; >x> \; b.put(x) \;\; \gg$

$\quad\quad if \;\; x :> 0 \; then \;\; in'.put(x-1) \; else \;\; signal \; \gg read()$

$\quad def \;\; write() = \; b.get() \;\; >x>$

$\quad\quad if \;\; x = 0 \; then \;\; out.put(1)$

$\quad\quad else \;\; (out'.get() \;\; >y> \; out.put(x * y)) \;\; \gg write()$

$read() \; | \; write()$

Program for fac

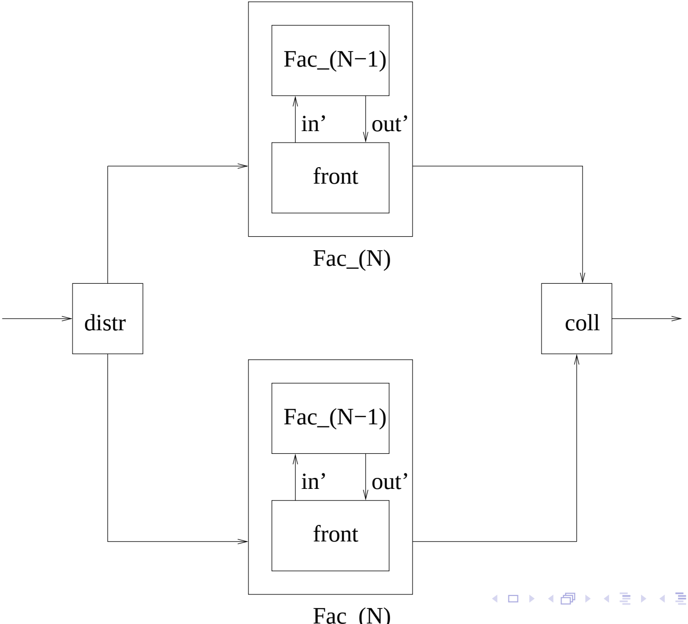
$$\mathit{def}\ \ \mathit{fac}(0, \mathit{in}, \mathit{out}) =$$
$$\quad \mathit{in}.\mathit{get}() \gg \mathit{out}.\mathit{put}(1) \gg \mathit{fac}(0, \mathit{in}, \mathit{out})$$

$$\mathit{def}\ \ \mathit{fac}(N, \mathit{in}, \mathit{out}) =$$
$$\quad \mathit{val}\ \ (\mathit{in}', \mathit{out}') = (\mathit{Channel}(), \mathit{Channel}())$$

$$\quad \mathit{def}\ \ \mathit{front}() = \cdots$$

$$\mathit{front}()\ |\ \mathit{fac}(N-1, \mathit{in}', \mathit{out}')$$

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:

$$sum([\,]) = 0,$$
$$sum(x : xs) = x + sum(xs)$$

$$length([\,]) = 0,$$
$$length(x : xs) = 1 + length(xs)$$

$$mean(xs) = sum(xs)/length(xs)$$

$$var([\,]) = 0,$$
$$var(xs) = mean(map(square, xs)) - mean(xs) ** 2$$

- 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.

Packet Reassembly Using Sequence Numbers

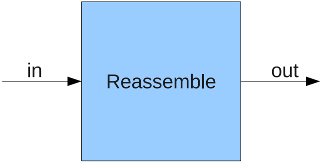


Figure: Packet Reassembler

- 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

$def \;\; reassembly(read, write, d) =$ – d must be positive
$\quad val \;\; ch = \; Table(d, lambda(\_) = Channel())$

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

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

$input() \mid output(0)$ – Goal expression

An Example Program: Broadcast

- Digital radio station has a list of subscribed listeners
- Broadcasts a message on dedicated channels to each one
- New listeners can be added

$$\textit{def}\ \ \textit{class Broadcast}(\textit{source}) =$$
$$\quad \textit{val listeners} = \textit{Ref}([\,])$$

$$\textit{def}\ \ \textit{addListener}(\textit{ch}) =$$
$$\quad \textit{listeners}?\ \ >\!\textit{fs}\!>\ \textit{listeners} := \textit{ch} : \textit{fs}$$

{- The ongoing computation of a broadcast -}

$$\textit{rep}(\textit{source})\ \ >\!\textit{item}\!>\ \textit{each}(\textit{listeners}?)\ \ >\!\textit{sink}\!>\ \textit{sink.put}(\textit{item})$$

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*().

Implementation Strategy

- Each instance of the stopwatch creates a new clock, starting at time 0.
- Maintains two Ref variables:
  laststart: clock value when the last start() was executed,
  timeshown: stopwatch value when the last halt() was executed.
- Initially, both variable values are 0.

Stopwatch Program

$def\ \ class\ Stopwatch() =$
$\quad val\ clk = Rclock()$
$\quad val\ (timeshown, laststart) = (Ref(0), Ref(0))$

$\quad def\ \ start() =\ \ laststart := clk.time()$

$\quad def\ \ halt() =$
$\quad timeshown := timeshown? + (clk.time() - laststart?) \gg$
$\quad timeshown?$

{- The ongoing computation of stopwatch -} $stop$

Stopwatch: Illegal starts and halts

- *start*() on a running watch has no effect. Publishes signal.
- *halt*() on a stopped watch has no effect. Publishes last value.
- *isrunning*() publishes true if and only if the stopwatch is running.
- Use a Ref variable to record if the stopwatch is running.

Stopwatch: Illegal starts and halts

```
def class Stopwatch() =
  val clk = Clock()
  val (timeshown, laststart) = (Ref(0), Ref(0))
  val running = Ref(false)

  def start() = if running? then signal
    else (running := true ≫ laststart := clk())

  def halt() =
    if running? then
      (timeshown? + (clk() − laststart?) >v>
      timeshown := v ≫ running := false ≫ v)
    else timeshown?

  def isrunning() = running?
  stop
```

Application: Measure running time of a site

$def\ \ class\ profile(f) =$
$\quad val\ \ sw = \ Stopwatch()$

$\quad def\ \ runningtime() = \ sw.start() \gg f() \gg sw.halt()$

$\quad stop$

-- 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 $(\mathit{true}, \mathit{response\ time})$, or $(\mathit{false}, \_)$ 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

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

$def\ \ game() =$

$\quad val\ \ v = \ Random(10)$ – v is the seed for one game

$val\ \ (b, w) =$

$\quad Rwait(id) \gg sw.reset() \gg rand\_seq() > x > Println(x) \gg$

$\quad Ift(x = v) \gg sw.start() \gg stop$

$\mid$ *Prompt*("Press ENTER for SEED "+v) $\gg$

$\quad sw.isrunning() > b > sw.pause() > w > (b, w)$

$if\ \ b\ then$ – Goal expression of $game()$

$\quad$("Your response time = " + w + " milliseconds.")

$else$ ("You jumped the gun.")

$game()$

Single alarm clock

Let $salarm$ be a single alarm clock.

- At any time at most one alarm can be set.
  A new alarm may be set after a previous alarm expires or is cancelled.

- $salarm.set(t)$ returns a signal after time t unless cancelled.
  The call blocks if alarm is already set or subsequently cancelled.

- $salarm.cancel()$ cancels the alarm and returns signal.
  Just returns a signal if no alarm has been set.
  This call is non-blocking.

Implementation Strategy for single alarm clock

- Ref variable *aset* shows if the alarm has been set.
- Semaphore *cancelled* is used to signal cancellation.
- Consider a scenario:
  An alarm is set for 100ms and cancelled at 50ms.
  Later, another alarm is set at 80ms to go off 40 ms later.
  The first alarm should not ring at 100ms
  (the thread must be pruned).

Implementation of Single alarm clock

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{class\ Alarm}() =\\
\quad \mathit{val}\ \ \mathit{aset} = \ \mathit{Ref}(\mathit{false})\\
\quad \mathit{val}\ \ \mathit{cancelled} = \ \mathit{Semaphore}(0)\\
\\
\quad \mathit{def}\ \ \mathit{cancel}() = \ \mathit{if}\ (\mathit{aset}?)\ \mathit{then}\ \mathit{cancelled.release}()\ \mathit{else}\ \mathit{signal}\\
\\
\quad \mathit{def}\ \ \mathit{set}(t) =\\
\qquad \mathit{Iff}(\mathit{aset}?) \gg \mathit{aset} := \mathit{true}\ \ \gg\\
\qquad (\mathit{val}\ \ b = \ \mathit{Rwait}(t) \gg \mathit{true} \mid \mathit{cancelled.acquire}() \gg \mathit{false}\\
\qquad\quad b \gg \mathit{aset} := \mathit{false} \gg \mathit{Ift}(b)\\
\qquad)\\
\\
\quad \mathit{stop}
\end{array}
$$

Clock with Multiple Alarm Setting

- Set an alarm with an id for a given time.
- Cancel an alarm (by its id) that has been set.
- A set alarm returns a signal unless it gets cancelled.
- An id can be reused.

Multiple Alarm Setting API

- Let $malarm$ be a multi-alarm clock in which n alarms may be simultaneously set.
- $malarm.set(i, t)$ returns a signal after time t unless cancelled. The call blocks if alarm is already set or later cancelled.
- $malarm.cancel(i)$ cancels the alarm with id i and returns signal. Just return a signal if no such id has been set. This call is non-blocking.
- A new alarm with some id can be set after the previous alarm with the same id expires.

Implementation of Multi-alarm clock

$$
\begin{aligned}
&\mathit{def}\ \ \mathit{class\ Multialarm}(n) = \\
&\quad \mathit{val}\ \ \mathit{alarmlist} = \ \mathit{Table}(n, \mathit{lambda}(\_) = \mathit{Alarm}()) \\
&\quad \mathit{def}\ \ \mathit{set}(i, t) = \ \mathit{alarmlist}(i).\mathit{set}(t) \\
&\quad \mathit{def}\ \ \mathit{cancel}(i) = \ \mathit{alarmlist}(i).\mathit{cancel}() \\
&\quad \mathit{stop}
\end{aligned}
$$

Testing Multialarm

$val\ \ m = \ Multialarm(5)$

$\ \ m.set(1, 500) \quad \gg$ "first alarm"
$|\ m.set(2, 100) \quad \gg$ "second alarm"
$|\ Rwait(400) \quad \gg m.cancel(1) \gg$ "first cancelled"
$|\ m.cancel(3) \quad \gg$ "No third alarm has been set"

- - - - - - - - - - - Output
"No third alarm has been set"
"second alarm"
"first cancelled"

Using Web services: Spellcheck a list of words

$$include\ "net.inc"$$

$$def\ \ spellCheck([\,]) = \ stop$$

$$\begin{aligned} def\ \ & spellCheck(word : words) = \\ & \quad GoogleSpellUnofficial(word)\ \ >sugg>\ (word, sugg) \\ & \ | \ spellCheck(words) \end{aligned}$$

$$spellCheck(["plese", "thereee", "Antiqu"])$$

Simulation as Concurrent Programming

- A simulation description is a real-time concurrent program.
- The concurrent program includes physical entities and their interactions.
- The concurrent program specifies time intervals for the activities.

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 $Succ()$

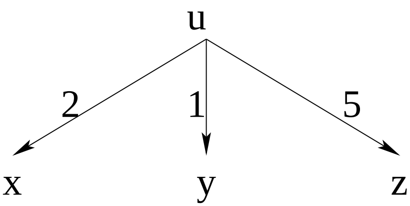


Figure: Graph Structure

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

Algorithm

$def \quad eval(u, t) =$ 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*

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.)

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

{- *Goal* :-} $\quad \mathit{eval}(\mathit{source}, 0) \mid \mathit{sink}?$

- 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.

Virtual Timer

Methods:

$Vwait(t)$	Returns a signal after t virtual time units.
$Vtime()$	Returns the current value of the virtual timer.

Virtual timer Properties

- Virtual timer value is monotonic.
- $Vwait(t)$ consumes exactly t units of virtual time.
- A step is started as soon as possible in virtual time.
- Virtual timer is advanced only if there can be no other activity.

Implementing virtual timer

Data structures:

- n: current value of $Vtime()$, initially $n = 0$.
- q: queue of calls to $Vwait()$ whose responses are pending.

At run time:

- A call to $Vtime()$ immediately responds with n.
- A call to $Vwait(t)$ is assigned rank $n + t$ and queued.
- Progress: If the program is stuck, then:

  remove the item with the lowest rank r from q,
  set $n := r$,
  respond with a signal to the corresponding call to $Vwait()$.

Examples

- $Rwait(10) \mid Ltimer(2)$
  Should logical timer be advanced with passage of real time?

- $Rwait(10) \gg c.put(5) \mid Ltimer(2)$
  Does $Rwait(10) \gg c.put(5)$ consume logical time?

- $c.get() \mid Ltimer(2) \gg c.put(5)$
  What are the values of $Ltimer.time()$ before and after $c.get()$?

- $stop \mid Ltimer(2)$
  Can the logical timer be advanced?

- $Google() \mid Ltimer(2)$
  Advance logical timer while waiting for $Google()$ to respond?
  What if $Google()$ never responds?

Simulation: Bank

- Bank with two tellers and one queue for customers.
- Customers generated by a *source* process.
- When free, a teller serves the first customer in the queue.
- Service times vary for customers.
- Determine
  - Average wait time for a customer.
  - Queue length distribution.
  - Average idle time for a teller.

Structure of bounded simulation

Run the simulation for *simtime*.
Below, $Bank()$ never publishes .

$$val\ \ z = \ Bank() \mid Vwait(simtime)$$

$$z \ \gg \ Stats()$$

Description of Bank

$$
\begin{array}{lcl}
\textit{def Bank}() & = & (\textit{Customers}() \mid \textit{Teller}() \mid \textit{Teller}()) \gg \textit{stop} \\
\textit{def Customers}() & = & \textit{Source}() \;>c>\; \textit{enter}(c) \\
\textit{def Teller}() & = & \textit{next}() \;>c> \\
 & & \textit{Vwait}(c.\textit{ServTime}) \gg \\
 & & \textit{Teller}() \\
\textit{def enter}(c) & = & q.\textit{put}(c) \\
\textit{def next}() & = & q.\textit{get}()
\end{array}
$$

Fast Food Restaurant

- Restaurant with one cashier, two cooking stations and one queue for customers.
- Customers generated by a *source* process.
- When free, cashier serves the first customer in the queue.
- Cashier service times vary for customers.
- Cashier places the order in another queue for the cooking stations.
- Each order has 3 parts: main entree, side dish, drink
- A cooking station processes parts of an order in parallel.

Goal Expression for Restaurant Simulation

$val\ \ z = \ Restaurant() \mid Vwait(simtime)$

$z \ \gg \ Stats()$

Description of Restaurant

$$
\begin{array}{lcl}
\textit{def Restaurant}() & = & (\textit{Customers}() \mid \textit{Cashier}() \mid \textit{Cook}() \mid \textit{Cook}()) \gg \textit{stop} \\
\\
\textit{def Customers}() & = & \textit{Source}() \;>c>\; \textit{enter}(c) \\
\\
\textit{def Cashier}() & = & \textit{next}() \;>c> \\
 & & \textit{Vwait}(c.\textit{ringupTime}) \gg \\
 & & \textit{orders.put}(c.\textit{order}) \gg \\
 & & \textit{Cashier}() \\
\\
\textit{def Cook}() & = & \textit{orders.get}() \;>\textit{order}> \\
 & & (\\
 & & \quad \textit{prepTime}(\textit{order.entree}) \;>t>\; \textit{Vwait}(t), \\
 & & \quad \textit{prepTime}(\textit{order.side}) \;>t>\; \textit{Vwait}(t), \\
 & & \quad \textit{prepTime}(\textit{order.drink}) \;>t>\; \textit{Vwait}(t) \\
 & &) \gg \textit{Cook}() \\
\\
\textit{def enter}(c) & = & q.\textit{put}(c) \\
\textit{def next}() & = & q.\textit{get}()
\end{array}
$$

Collecting Statistics: waiting time

Change

$$\begin{array}{lcl} def\ enter(c) & = & q.put(c) \\ def\ next() & = & q.get() \end{array}$$

to

$$def\ enter(c) \quad = \quad Vtime() \ >s> \ q.put(c, s)$$

$$\begin{array}{lcl} def\ next() & = & q.get() \ >(c,t)> \\ & & \quad Vtime() \ >s> \\ & & \quad reportWait(s - t) \ \gg \\ & & \quad c \end{array}$$

Histogram: Queue length

- Create $N+1$ stopwatches, $sw[0..N]$, at the beginning of simulation.
- Final value of $sw[i]$, $0 \leq i < N$, is the duration for which the queue length has been i.
- $sw[N]$ is the duration for which the queue length is at least N.
- On adding an item to queue of length i, $0 \leq i < N$, do

$$sw[i].stop \mid sw[i+1].start$$

- After removing an item if the queue length is i, $0 \leq i < N$, do

$$sw[i].start \mid sw[i+1].stop$$

Simulation Layering

- A simulation is written a set of layers.
- Lowest layer represents the abstraction of the physical system.
- Next layer may collect statistics, by monitoring the layer below it.
- Further layers may produce reports and animations from the statistics.