Some Cute Programs

Jayadev Misra

Department of Computer Science
University of Texas at Austin

http://orc.csres.utexas.edu

July 16 – 20, 2012
WG 2.3 Meeting, Seattle
Orc Basics

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

No notion of data type, thread, process, channel, synchronization, parallelism  · · ·

New concepts are programmed using new sites.
Structure of Orc Expression

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

- **Composition** of two Orc expressions:

  \[
  \begin{align*}
  \text{do } f \text{ and } g \text{ in parallel} & \quad f \mathbin{\|} g & \text{Symmetric composition} \\
  \text{for all } x \text{ from } f \text{ do } g & \quad f \mathbin{\succ} x \mathbin{\succ} g & \text{Sequential composition} \\
  \text{for some } x \text{ from } g \text{ do } f & \quad f \mathbin{\prec} x \mathbin{\prec} g & \text{Pruning} \\
  \text{if } f \text{ halts without publishing do } g & \quad f \mathbin{;} g & \text{Otherwise}
  \end{align*}
  \]
Structure of Orc Expression

- **Simple**: just a site call, $\text{CNN}(d)$
  Publishes the value returned by the site.

- **Composition** of two Orc expressions:

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

  \[
  \text{for all } x \text{ from } f \text{ do } g \quad \quad \quad f \triangleright x \triangleright g \quad \quad \quad \text{Sequential composition}
  \]

  \[
  \text{for some } x \text{ from } g \text{ do } f \quad \quad \quad f \triangleleft x \triangleleft g \quad \quad \quad \text{Pruning}
  \]

  \[
  \text{if } f \text{ halts without publishing do } g \quad \quad \quad f ; g \quad \quad \quad \text{Otherwise}
  \]
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
    \[ f \mid g \]
  - for all \( x \) from \( f \) do \( g \)
    \[ f >x> g \]
  - for some \( x \) from \( g \) do \( f \)
    \[ f <x< g \]
  - if \( f \) halts without publishing do \( g \)
    \[ f ; g \]

Symmetric composition
Sequential composition
Pruning
Otherwise
Structure of Orc Expression

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

- **Composition** of two Orc expressions:

  - \( f \) and \( g \) in parallel
    \[ f \mid g \]
    Symmetric composition
  
  - for all \( x \) from \( f \) do \( g \)
    \[ f > x > g \]
    Sequential composition
  
  - 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: $f \gg x \gg g$

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

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

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

Notation: $f \gg g$ for $f \gg x \gg g$, if $x$ is unused in $g$. 
Schematic of Sequential composition

Figure: Schematic of $f \succ x \succ g$
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 $g$. 
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.
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.
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

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

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

- \textit{stop}: never responds. Same as \textit{Ift}(false) or \textit{Iff}(true).

- \textit{signal}: returns a signal immediately.
  Same as \textit{Ift}(true) or \textit{Iff}(false).
Example of a Definition: Metronome

Publish a signal every unit.

\[
\text{def Metronome}() = \begin{cases} 
\text{signal} & \text{S} \\
\left( \text{Rwait}(1) \gg \text{Metronome}() \right) & \text{R}
\end{cases}
\]
Unending string of Random digits

\[\text{Metronome()} \gg \text{Random}(10) \text{ -- one every second}\]

\[
\text{def } \text{rand_seq}() = \\
\text{Random}(10) \mid \text{Rwait}(dd) \gg \text{rand_seq}()
\]
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”

\( b <b< (M() \mid N()) \) – “or”

\( M() \; ; \; N() \) – “or” with helpful \( M \)

\( (M() \gg true \; ; \; false) >b> Iff(b) \) – “not” with helpful \( M \)
Orc Language

- **Data Types**: Number, Boolean, String, with Java operators

- **Conditional Expression**: `if b then f else g`

- **Data structures**: Tuple, List, Record

- **Pattern Matching; Clausal Definition**

- **Function Closure**

- **Comingling functional and Orc expressions**

- **Class for active objects**
Implicit Concurrency

- An experiment tosses two dice. Experiment is a success if and only if sum of the two dice thrown is 7.
- \( \text{exp}(n) \) runs \( n \) experiments and reports the number of successes.

\[
def \text{toss}() = \text{Random}(6) + 1 \\
\text{-- \text{toss} returns a random number between 1 and 6}
\]

\[
def \text{exp}(0) = 0 \\
def \text{exp}(n) = \text{exp}(n - 1) \\
\quad + (\text{if} \ \text{toss}() + \text{toss}() = 7 \ \text{then} \ 1 \ \text{else} \ 0)
\]
Translation of the dice throw program

```python
def toss() = add(x, 1) < x < Random(6)

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

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

Note: 2n parallel calls to `toss()`.
Deflation

- Expression $C(..., e, ..)$,

- single value expected at $e$

- translate to $C(..., x, ..) \prec x \prec e$ where $x$ is fresh

- applicable hierarchically.

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

$(\text{Times}(x, y) \prec x \prec (1 | 2)) \prec y \prec (10 | 100)$, or

$\text{Times}(x, y) \prec x \prec (1 | 2) \prec y \prec (10 | 100)$
Pattern Matching, clausal definition

\[
\text{type } \text{Tree} = \text{Node} (\text{Tree}, \text{Tree}) \mid \text{Leaf} () \mid \text{NonTree} ()
\]

\[
def \text{tc} (_, []) = \text{NonTree} ()
\]

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

\[
def \text{tc} ([], v : \text{right}) = \text{tc} ([v], \text{right})
\]

\[
def \text{tc} ((u, t) : \text{left}, (v, t') : \text{right}) =
\]
\[
\text{if } u = v \text{ then } \text{tc} (\text{left}, (v - 1, \text{Node} (t, t')) : \text{right})
\]
\[
\text{else } \text{tc} ((v, t') : (u, t) : \text{left}, \text{right})
\]
def circle =

val pi = 3.1416

def perim(r) = 2 * pi * r

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

(perim, area)
Examples

- Combinatorial
- Mutable store manipulation
- Synchronization, Communication
List map

def parmap(_, []) = []

def parmap(f, x : xs) = f(x) : parmap(f, xs)
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.

```python
def bin() =
    []
    | bin() >>= (0 : xs | 1 : xs)
```
Subset Sum

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

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

\[
\begin{align*}
\text{def } \text{parsum}(0, []) &= [] \\
\text{def } \text{parsum}(n, []) &= \text{stop} \\
\text{def } \text{parsum}(n, x : xs) &= \text{parsum}(n - x, xs) > ys > x : ys \mid \text{parsum}(n, xs)
\end{align*}
\]
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.

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

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

\[
def \text{seqsum}(n, x: xs) = \\
x : \text{seqsum}(n - x, xs) ; \text{seqsum}(n, xs)
\]
Publish the first sublist of $xs$ that sums to $n$.

Run the searches concurrently.

```python
def parseqsum(0, []) = []
def parseqsum(n, []) = stop
def parseqsum(n, x : xs) =
    (p ; q)
    <p< x : parseqsum(n − x, xs)
    <q< parseqsum(n, xs)
```

Note: Neither search in the last clause may succeed.
Fold on a non-empty list

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

\[
\text{def} \quad \text{fold}([], [x]) = x
\]

\[
\text{def} \quad \text{fold}(f, x : xs) = f(x, \text{fold}(xs))
\]
Associative fold on a non-empty list

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

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

map and associative fold:  \textit{map\_afold}
A channel has two methods: *put* and *get*.

\( \text{chFold}(c, n) \) folds the first \( n \) items of channel \( c \) and publishes.

\[
\text{def} \quad \text{chFold}(c, 1) = c.\text{get}()
\]

\[
\text{def} \quad \text{chFold}(c, n) = f(\text{chFold}(c, n/2), \text{chFold}(c, n - n/2))
\]
Associative commutative fold over a channel

\[
\text{def } \text{cfold}(c, n) = \\
\text{def } \text{threads}(0) = \text{stop} \\
\text{def } \text{threads}(k) = \\
\quad \text{threads}(k - 1) \\
\quad | \quad c\text{.put}(f(c\text{.get}(), c\text{.get}())) \gg \text{stop} \\
\text{threads}(n - 1) ; c\text{.get}() \\
\]

- if \( n \) is strictly more than \( k \), \( \text{threads}(k) \) terminates.
- at its termination the channel contains \( n - k \) items whose fold yields the desired result.
Mutable Store Manipulation

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
Channel()    Unbounded (asynchronous) channel

\[
\text{Ref}(3) \triangleright r \triangleright r.\text{write}(5) \triangleright r.\text{read}(), \text{ or } \text{Ref}(3) \triangleright r \triangleright r := 5 \triangleright r\
\]

\[
\text{Cell()} \triangleright r \triangleright (r.\text{write}(5) | r.\text{read}()), \text{ or } \text{Cell()} \triangleright r \triangleright r := 5 | r\
\]

\[
\text{Array}(3) \triangleright a \triangleright a(0) := \text{true} \triangleright a(1)\
\]

\[
\text{Channel()} \triangleright ch \triangleright (ch.\text{get}() | ch.\text{put}(3) \triangleright \text{stop})\
\]
Quicksort

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

(lr(s + 1), rl(t - 1)) > (s', t') >
(if (s' < t') then swap(a(s'), a(t')) \gg part(p, s', t')
else t')

if v - u > 1 then
    part(a(u)?, u, v) > m >
    swap(a(u), a(m)) \gg
    (segmentsort(u, m), segmentsort(m + 1, v)) \gg signal
else signal
segmentsort(0, a.length?)
```
Sequential Breadth-First Traversal of a Graph

\( N \) nodes in a graph,

\( \text{root} \) a specified node,

\( \text{succ}(x) \) is the list of successors of \( x \),

Publish the \( \text{parent} \) of each node in Breadth-First Traversal.

\[
\text{def } \text{bfs}(N, \text{root}, \text{succ}) = \\
\text{val } \text{parent} = \text{Table}(N, \lambda(_) = \text{Cell}())
\]

\( - \) \text{bfs}' is \( \text{bfs} \) on a list of nodes

\[
\text{def } \text{bfs}'([[]]) = \text{signal} \\
\text{def } \text{bfs}'(x : xs) = \text{bfs}'(\text{append}(xs, \text{expand}(x)))
\]

\( \text{parent}(\text{root}) := N \gg \text{bfs}'([\text{root}]) \gg \text{parent} \)
def expand(x) =

  - expand′(x, ys), ys successors of \( x \) yet to be scanned

  def expand′(x, []) = []
  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

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

def expand(x) =
    def expand'(x, []) = []
    def expand'(x, z : zs) =
        parent(z) := x $\Rightarrow$ z : expand'(x, zs) ; expand'(x, zs)
    expand'(x, succ(x)) -- Goal of expand

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

parent(root) := N $\Rightarrow$ bfs'([root]) $\Rightarrow$ parent
```
Concurrent Breadth-First Traversal

```python
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 $f()$.

$$
\text{val } \text{done } = \text{Cell}() \\
\text{val } \text{res } = \text{Cell}() \\
\text{def } \text{memof}() = \\
\text{res? } \ll (\text{done } := \text{signal} ) \gg \text{res } := f()$$
Memoization of Fibonacci

\[ val \ N = 100 \]
\[ val \ done = Table(N + 1, lambda(_): Cell()) \]
\[ val \ res = Table(N + 1, lambda(_): Cell()) \]

\[ def \ mfib(0):= 0 \]
\[ def \ mfib(1):= 1 \]
\[ def \ mfib(i) = \]
\[ \quad res(i) \Leftarrow \]
\[ \quad \text{(done}(i) ::= \text{signal} \Rightarrow res(i) ::= mfib(i - 1) + mfib(i - 2)) \]
Synchronization, Communication

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

Semaphore(1) >s> s.acquire() ⇒ r := 5 ⇒ s.release()

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

Counter() >ctr> (ctr.inc() ⇒ ctr.onZero() | Rwait(10) ⇒ ctr.dec())
Rendezvous

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

    def put(x) = s.acquire() \> w.put(x)
    def get() = s.release() \> w.get()

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

  def put() = s.acquire() >> t.release()
  def get() = s.release() >> t.acquire()

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

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

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

def endread() = na.dec()

def endwrite() = na.dec()
def manager() = grant(req.get()) >>= manager()

def grant((true, s)) = na.inc() >>= s.release() -- Reader

def grant((false, s)) = -- Writer
    na.onZero() >>= na.inc() >>= s.release() >>= na.onZero()
Reader-Writer; Using 2 semaphores

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

Keep count of the number of waiting readers and writers.

Use coin toss to choose a reader or writer, instead of looking up in the queue.
Packet Reassembly Using Sequence Numbers

- Packet with sequence number $i$ is at position $p_i$ in the input channel.

- Given: $|i - p_i| \leq k$, for some positive integer $k$.

- Then $p_i \leq i + k \leq p_{i+2	imes k}$. Let $d = 2 \times k$. 
def reassembly(read, write, d) = – d must be positive
val ch = Table(d, lambda(_)=Channel())

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

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

input() | output(0) – Goal expression

{- With Multiple Readers -} read() | read() | write(0)
Response Game

\begin{align*}
\text{val } sw &= \text{Stopwatch}() \\
\text{val } (id, dd) &= (3000, 100) \quad \text{– initial delay, digit delay} \\
\text{def } \text{rand\_seq}() &= \quad \text{– Publish a random sequence of digits} \\
&\quad \text{Random}(10) \mid \text{Rwait}(dd) \gg \text{rand\_seq}() \\
\text{def } \text{game}() = \\
&\quad \text{val } v = \text{Random}(10) \quad \text{– } v \text{ is the seed for one game} \\
&\quad \text{val } (b, w) = \\
&\quad \quad \text{Rwait}(id) \gg \text{sw.reset()} \gg \text{rand\_seq()} >x> \text{Println}(x) \gg \\
&\quad \quad \text{Ift}(x = v) \gg \text{sw.start()} \gg \text{stop} \\
&\quad \mid \text{Prompt( "Press ENTER for SEED "} + v \text{ )} \gg \\
&\quad \text{sw.isrunning()} >b> \text{sw.halt()} >w> (b, w) \\
\text{if } b \text{ then } \quad \text{– Goal expression of } \text{game}() \\
&\quad ( \text{"Your response time = "} + w + \text{" milliseconds."} ) \\
\text{else } ( \text{"You jumped the gun."} ) \\
\text{game}() 
\end{align*}