Structured Wide-Area Programming: Orc Abstraction, Class

Jayadev Misra

Department of Computer Science
University of Texas at Austin

http://orc.csres.utexas.edu

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

```
def class Stack(n) =
    val store = Table(n, lambda(_) = Ref())
    val len = Ref(0)

    def push(x) =
        Ift(len? <: n) >> store(len?) := x >> len := len? + 1

    def pop() =
        Ift(len? :> 0) >> len := len? - 1 >> store(len?)?

  {- class Goal -} stop
```

---------- Test

```
val st = Stack(5)
st.push(3) >> st.push(5) >> st.pop() >> 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 function 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 function definition apply except:
  - Publications of class goal expression are ignored,
  - Each method (function) publishes at most once (to mimic a site),
  - Class calls are strict (function 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 functions.

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)

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{class}\ \mathit{Matrix}((\mathit{row},\mathit{row}'),(\mathit{col},\mathit{col}')) = \\
\quad \mathit{val}\ \ \mathit{mat} = \ \mathit{Array}((\mathit{row}' - \mathit{row} + 1) * (\mathit{col}' - \mathit{col} + 1)) \\
\quad \mathit{def}\ \ \mathit{access}(i,j) = \ \mathit{mat}((i - \mathit{row}) * (\mathit{col}' - \mathit{col} + 1) + j) \\
\quad \mathit{stop}
\end{array}
$$

---------- 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

$$\begin{array}{l}
\mathit{def}\ \ \mathit{class}\ \mathit{CMatrix}((\mathit{row},\mathit{row}'),(\mathit{col},\mathit{col}'),\mathit{cap}) = \\
\quad \mathit{val}\ \ \mathit{mat} = \ \mathit{Table}((\mathit{row}' - \mathit{row} + 1) * (\mathit{col}' - \mathit{col} + 1), \mathit{cap}) \\
\quad \mathit{def}\ \ \mathit{access}(i,j) = \ \mathit{mat}((i - \mathit{row}) * (\mathit{col}' - \mathit{col} + 1) + j) \\
\quad \mathit{stop}
\end{array}$$

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

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

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

Create a new site: Cell using Semaphore and Ref

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{class}\ \mathit{Cell}() = \\
\quad \mathit{val}\ \ s = \ \mathit{Semaphore}(1) \\
\quad \mathit{val}\ \ r = \ \mathit{Ref}\,() \\
\quad \mathit{def}\ \ \mathit{write}(v) = \ s.\mathit{acquire}() \gg r := v \\
\quad \mathit{def}\ \ \mathit{read}() = \ r?\quad \text{- -}\quad r?\ \text{blocks until}\ \ r\ \text{has been written} \\
\quad \mathit{stop}
\end{array}
$$

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.

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{class}\ \mathit{BChannel}(n) = \\
\quad \mathit{val}\ \ b = \ \mathit{Channel}() \\
\quad \mathit{val}\ \ p = \ \mathit{Semaphore}(n) \\
\\
\quad \mathit{def}\ \ \mathit{put}(x) = \ p.\mathit{acquire}() \gg b.\mathit{put}(x) \\
\\
\quad \mathit{def}\ \ \mathit{get}() = \ b.\mathit{get}() \ \ >x> \ p.\mathit{release}() \gg x \\
\\
\quad \mathit{stop}
\end{array}
$$

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

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

Memoization

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

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

```
val res = Cell()
val s = Semaphore(1)
def memo() =
    val z = res? | s.acquire() ≫ res := f() ≫ stop
    z
```

Note: Concurrent calls handled correctly.

Memoize an argument function using Class

```
def class Memo(f) =
  val res = Cell()
  val s = Semaphore(1)

  def memo() =
    val z = res? | s.acquire() >> res := f() >> stop
      z

  stop
```

— Usage

```
val prandom = Memo(lambda() = Random(20)).memo
prandom() | prandom() | 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() ≫
       u.put(x) ≫ v.get() >y>
       sem.release() ≫ y

  def read() = u.get()

  def write(x) = v.put(x)

  stop
```

Rendezvous

- n parties need to synchronize, $n \geq 2$.
- For 2 parties: one calls $send()$, the other $recv()$.
- A call is blocked until both calls are ready.
  Then both calls return signals.

Implementation strategy for 2-party pure Synchronization

- Use two semaphores, s and r.
- $send()$ acquires s and releases r, in this order.
- $recv()$ releases s and acquires r, in this order.

Rendezvous; 2-party, Pure Synchronization

```
def class Rendezvous2() =
  val (s, r) = (Semaphore(0), Semaphore(0))

  def send() = s.acquire() ≫ r.release()
  def recv() = s.release() ≫ r.acquire()

  stop

---------- Test
  val group1 = Rendezvous2()
  val group2 = Rendezvous2()

  group1.send() | Rwait(1000) ≫ group2.recv()
| group2.send() ≫ group1.recv()
```

Rendezvous with data transfer

```
def class Rendezvous2d() =
  val (s, data) = (Semaphore(0), Channel())

  def send(x) = s.acquire() ≫ data.put(x)
  def recv() =  s.release() ≫ data.get()

  stop

---------- Test
  val group1 = Rendezvous2d()
  val group2 = Rendezvous2d()

  group1.send(3) | Rwait(1000) ≫ group2.recv()
| group2.send(5) ≫ group1.recv()
```

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

- Each party is a client.
- A server receives input data.
  After receiving all n pieces, it applies the given function.
  It places the results so that the clients can read them.
- Use a Table of classes of type $csi()$:
  $req(x)$: for client to supply data x
  $read()$: for server to read
  $write(v)$: for server to write.

Program

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{class}\ \mathit{Rendezvous}(n, f) = \\
\quad \mathit{val}\ \ b = \ \mathit{Table}(n, \mathit{lambda}(\_) = \mathit{csi}()\ ) \\
\\
\quad \mathit{def}\ \ \mathit{go}(i, x) = \ b(i).\mathit{req}(x) \\
\\
\quad \mathit{def}\ \ \mathit{collect}(vl, 0) = \ vl \\
\quad \mathit{def}\ \ \mathit{collect}(vl, i) = \ b(n-i).\mathit{read}()\ \ >v>\ v : \mathit{collect}(vl, i-1) \\
\\
\quad \mathit{def}\ \ \mathit{distribute}(\_, 0) = \ \mathit{signal} \\
\quad \mathit{def}\ \ \mathit{distribute}(v : vl, i) = \ b(n-i).\mathit{write}(v) \gg \mathit{distribute}(vl, i-1) \\
\\
\quad \mathit{def}\ \ \mathit{manager}() = \\
\qquad \mathit{collect}([\,], n)\ \ >vl>\ \mathit{distribute}(f(vl), n) \gg \mathit{manager}() \\
\\
\quad \mathit{manager}()
\end{array}
$$

Test

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

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

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

- - - - - - - - - - Output

"0 gets 1"

"1 gets 4"

"2 gets 0"

Barrier Synchronization; A special case of Rendezvous

- 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 $barrier()$ after each phase, and waits to receive a $signal$ to execute its next phase.
- Rendezvous where input and output are both $signal$.

Typical Usage:

$def\ \ class\ BarrierSync(n) = \ \cdots$
$val\ \ barrier = \ BarrierSync(3).go$

---------- Test

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

Program: Barrier Synchronization

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{class}\ \mathit{BarrierSync}(n) = \\
\quad \mathit{val}\ \ (\mathit{in}, \mathit{out}) = \ (\mathit{Counter}(n), \mathit{Counter}(0)) \\
\\
\quad \mathit{def}\ \ \mathit{go}() = \ \mathit{in}.\mathit{dec}() \ \gg \ \mathit{out}.\mathit{dec}() \\
\\
\quad \mathit{def}\ \ \mathit{manager}() = \\
\qquad\quad \mathit{in}.\mathit{onZero}() \ \gg \ \mathit{out}.\mathit{reset}(n) \ \gg \\
\qquad\quad \mathit{out}.\mathit{onZero}() \ \gg \ \mathit{in}.\mathit{reset}(n) \ \gg \\
\qquad\quad \mathit{manager}() \\
\\
\quad \mathit{manager}()
\end{array}
$$

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 $s.release()$

Readers-Writers Program

```
def class ReaderWriter() =
  val (req, ctr) = (Channel(), Counter())

  def start(b) =
    val s = Semaphore(0)
    req.put((b, s)) ≫ s.acquire()

  def end() = ctr.dec()

  def manager() = req.get() >(b, s)> grant(b, s) ≫ manager()

  def grant(true, s) = ctr.inc() ≫ s.release() {- Reader -}
  def grant(false, s) = {- Writer -}
      ctr.onZero() ≫ ctr.inc() ≫ s.release() ≫ ctr.onZero()

  manager()
```

Note on Callback

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

Callback using one semaphore each for Readers and Writers

```
def class ReaderWriter() =
  val (req, ctr) = (Channel(), Counter())
  val (r, w) = (Semaphore(0), Semaphore(0))

  def start(true) = req.put(true) ≫ r.acquire()
  def start(false) = req.put(false) ≫ w.acquire()

  def end() = ctr.dec()

  def manager() = req.get() >b> grant(b) ≫ manager()

    def grant(true) = ctr.inc() ≫ r.release()
    def grant(false) =
        ctr.onZero() ≫ ctr.inc() ≫ w.release() ≫ ctr.onZero()

  manager()
```

Further refining the solution; exercise

- 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.

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

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{class\ Stopwatch}() = \\
\quad \mathit{val}\ \ \mathit{clk} = \ \mathit{Clock}() \\
\quad \mathit{val}\ \ (\mathit{timeshown}, \mathit{laststart}) = \ (\mathit{Ref}(0), \mathit{Ref}(0)) \\
\\
\quad \mathit{def}\ \ \mathit{start}() = \ \mathit{laststart} := \mathit{clk}() \\
\\
\quad \mathit{def}\ \ \mathit{halt}() = \\
\qquad \mathit{timeshown} := \mathit{timeshown}? + (\mathit{clk}() - \mathit{laststart}?) \gg \\
\qquad \mathit{timeshown}? \\
\quad \mathit{stop}
\end{array}
$$

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 function

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

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

  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 $(true, response\ time)$, or $(false, \_)$ if v has not appeared. Then end the game.

Response Time Game

```
val sw = Stopwatch()
val (id, dd) = (3000, 100) -- id, dd are delays

def rand_seq() = Random(10) | Rwait(dd) >> rand_seq()

def game() =                          -- run a game
  val v = Random(10)                  -- v is a random digit

  val (b, w) =
      Prompt( "Press ENTER/OK for " + v) >>
      sw.isrunning() >b> sw.halt() >w> (b, w)

    | Rwait(id) >> rand_seq() >x> Println(x) >>
      Ift(x = v) >> sw.start() >> stop

Println( "output =" + (b, w))
```

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

```
def class Alarm() =
  val aset = Ref(false)
  val cancelled = Semaphore(0)

  def cancel() = if (aset?) then cancelled.release() else signal

  def set(t) =
      Iff(aset?) ≫ aset := true ≫
      (val b = Rwait(t) ≫ true | cancelled.acquire() ≫ false
           b ≫ aset := false ≫ Ift(b)
      )

  stop
```

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

```
def class Multialarm(n) =
  val alarmlist = Table(n, lambda(_) = Alarm())

  def set(i, t) = alarmlist(i).set(t)

  def cancel(i) = alarmlist(i).cancel()

  stop
```

Testing Multialarm

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

$$\begin{array}{lll} \ \ m.set(1,500) & \gg & \text{"first alarm"} \\ \mid m.set(2,100) & \gg & \text{"second alarm"} \\ \mid Rwait(400) & \gg & m.cancel(1) \gg \ \text{"first cancelled"} \\ \mid m.cancel(3) & \gg & \text{"No third alarm has been set"} \end{array}$$

- - - - - - - - - - Output

"No third alarm has been set"

"second alarm"

"first cancelled"