Concurrent and Distributed Programming Patterns in Oz (VRH Chs 1,4,5,8,11)

Carlos Varela
RPI
March 19, 2007
Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

Overview
- Programming techniques and patterns
  - stream communication,
  - order-determining concurrency,
  - coroutines,
  - concurrent composition
  - soft real-time programming
- Active objects
- Decentralized (P2P) coordination
- Self-management

Streams
- A stream is a sequence of messages
- A stream is a First-In First-Out (FIFO) channel
- The producer augments the stream with new messages, and the consumer reads the messages, one by one.

Stream Communication

Stream Communication I
- The data-flow property of Oz easily enables writing threads that communicate through streams in a producer-consumer pattern.
- A stream is a list that is created incrementally by one thread (the producer) and subsequently consumed by one or more threads (the consumers).
- The consumers consume the same elements of the stream.
Stream communication patterns

- The producer, transducers, and the consumer can, in general, be described by certain program patterns
- We show various patterns

Example Producer

fun (Producer State)
  if (More State) then
    X = (Produce State) in
    X | {Produce (Transform State)}
  else nil end
end

- The definition of More, Produce, and Transform is problem dependent
- State could be multiple arguments
- The above definition is not a complete program!

Example Consumer

fun (Consumer State InStream)
case InStream of nil then {
  Final State}
[] X | RestInStream then
  NextState = {
    Consume X State}
in
  {Consumer NextState RestInStream}
end
end

- Final and Consume are problem dependent

Transducer Pattern 1

fun (Transducer State InStream)
case InStream of nil then nil
[] X | RestInStream then
  NextState = TX = (Transform X State)
in
  {Transducer NextState RestInStream}
end
end

- A transducer keeps its state in State, receives messages on InStream and sends messages on OutStream

Example Producer

fun {Generate N Limit}
  if N=<Limit then
    N | {Generate N+1 Limit}
  else
    nil
  end
end

- The State is the two arguments N and Limit
- The predicate More is the condition N=<Limit
- The predicate Produce is the identity function on N
- The Transform function (N,Limit) ⇒ (N+1,Limit)

Example Consumer

fun {Sum A Xs}
case Xs of nil then A
 || X | Xr then {Sum A+X Xr}
end
end

- The State is A
- Final is just the identity function on State
- Consumer takes X and State ⇒ X + State
Transducer Pattern 2

fun {Transducer State Instream}
case Instream
  of nil then nil
  | X RestInstream then
  if (Test X(State)) then
    NextState/IX = {Transform X State}
    TX | {Consumer NextState RestInstream}
  else
    {Consumer NextState RestInstream}
end
end
• A transducer keeps its state in State, receives messages on InStream and sends messages on OutStream

Example Transducer

fun {Filter Xs F}
case Xs
  of nil then nil
  | X| Xr then
  if {F X} then
    X| {Filter Xr F}
  else
    {Filter Xr F}
  end
end
end

Example call

local Xs Ys in
thread Xs = {Generate 2 100000} end
thread Ys = {Sieve Xs} end
thread for Y in Ys do {Show Y} end
end

Larger example:
The sieve of Eratosthenes

• Produces prime numbers
• It takes a stream 2...N, peaks off 2 from the rest of the stream
• Delivers the rest to the next sieve

Sieve

fun {Sieve Xs}
case Xs
  of nil then nil
  | X| Xr then
    Ys in
    thread Ys = {Filter Xr fun {$ Y} Y mod X = 0 end} end
    X | {Sieve Ys}
  end
end
• The program forks a filter thread on each sieve call

Limitation of eager stream processing Streams

• The producer might be much faster than the consumer
• This will produce a large intermediate stream that requires potentially unbounded memory storage
Solutions

- There are three alternatives:
  1. Play with the speed of the different threads, i.e. play with the scheduler to make the producer slower.
  2. Create a bounded buffer, say of size N, so that the producer waits automatically when the buffer is full.
  3. Use demand-driven approach, where the consumer activates the producer when it needs a new element (lazy evaluation).

- The last two approaches introduce the notion of flow-control between concurrent activities (very common).

Coroutines I

- Languages that do not support concurrent threads might instead support a notion called corouting.
- A coroutine is a nonpreemptive thread (sequence of instructions), there is no scheduler.
- Switching between threads is the programmer’s responsibility.

Co-routines, Time, Termination Detection, Futures, Priorities

Coroutines II, Comparison

- In concurrent computation one would like to handle time:
  - proc {Time.delay T} – The running thread suspends for T milliseconds.
  - proc {Time.alarm T U} – Immediately creates its own thread, and binds U to unit after T milliseconds.

Co-routines II, Comparison

- In concurrent computation one would like to handle time:
  - proc {Time.delay T} – The running thread suspends for T milliseconds.
  - proc {Time.alarm T U} – Immediately creates its own thread, and binds U to unit after T milliseconds.
**Example**

```
local proc {Ping N}
for I in 1..N do
  {Delay 500} {Browse ping}
end {Browse 'ping terminate'}
end

local proc {Pong N}
for I in 1..N do
  {Delay 600} {Browse pong}
end {Browse 'pong terminate'}
end
```

**Concurrent control abstraction**

- We have seen how threads are forked by "thread ... end"
- A natural question to ask is: how can we join threads?

```
fork join threads
```

**Termination detection**

- This is a special case of detecting termination of multiple threads, and making another thread wait on that event.
- The general scheme is quite easy because of dataflow variables:

```
thread (S1) X1 = unit end
thread (S2) X2 = X1 end...

thread (Sn) Xn = Xn-1 end
```

- When all threads terminate the variables X1 ... Xn will be merged together labeling a single box that contains the value unit.
- `{Wait Xn}` suspends the main thread until Xn is bound.

**Concurrent Composition**

```
conc S1 [] S2 [] ... Sn end
```

- Takes a single argument that is a list of nullary procedures.
- When it is executed, the procedures are forked concurrently. The next statement is executed only when all procedures in the list terminate.

**Example**

```
local proc {Conc1 Ps I O}
case Ps of P|Pr then
  M in thread {P} M = I end {Conc1 Pr M O}
end
end proc {Conc Ps}
X in {Conc1 Ps unit X} {Wait X} end
```

- This abstraction takes a list of zero-argument procedures and terminate after all these threads have terminated.
Futures

- A future is a read-only capability of a single-assignment variable. For example, to create a future of the variable \( X \) we perform the operation \( !X \) to create a future \( Y \):

\[
Y = !X
\]

- A thread trying to use the value of a future, e.g., using \( Y \), will suspend until the variable of the future, e.g., \( X \), gets bound.

- One way to execute a procedure lazily, i.e., in a demand-driven manner, is to use the operation \( \{ByNeed +P \?F\} \).

- \( ByNeed \) takes a zero-argument function \( P \), and returns a future \( F \). When a thread tries to access the value of \( F \), the function \( \{P\} \) is called, and its result is bound to \( F \).

- This allows us to perform demand-driven computations in a straightforward manner.

Example

- \( \text{declare} \ Y \{ByNeed \text{fun} \{X\} \text{end} Y\} \{Browse \ Y\} \)

- we will observe that \( Y \) becomes a future, i.e., we will see \( Y<\text{Future}> \) in the Browse.

- If we try to access the value of \( Y \), it will get bound to \( 1 \).

- One way to access \( Y \) is by perform the operation \( \{\text{Wait} \ Y\} \) which triggers the producing procedure.

Thread Priority and Real Time

- Try to run the program using the following statement:

\[
- \{\text{Sum} 0 \text{thread} \{\text{Generate} 0 10000000\} \text{end}\}
\]

- Switch on the panel and observe the memory behavior of the program.

- You will quickly notice that this program does not behave well.

- The reason has to do with the asynchronous message passing. If the producer sends messages, i.e., create new elements in the stream, in a faster rate than the consumer can consume, increasingly more buffering will be needed until the system starts to break down.

- One possible solution is to control experimentally the rate of thread execution so that the consumers get a larger time-slice than the producers do.

Priorities

- There are three priority levels:

  - high,
  - medium, and
  - low (the default)

- A priority level determines how often a runnable thread is allocated a time slice.

- In Oz, a high-priority thread cannot starve a low-priority one. Priority determines only how large a piece of the processor-cake a thread can get.

- Each thread has a unique name. To get the name of the current thread the procedure \( \text{Thread.this/1} \) is called.

- Having a reference to a thread, by using its name, enables operations on threads such as:

  - terminating a thread, or
  - raising an exception in a thread.

- Thread operations are defined in the standard module \( \text{Thread} \).

Thread Priorities

- Oz has three priority levels. The system procedure

\[
\{\text{Property.put priorities p(medium:Y high:X)}\}
\]

- Sets the processor-time ratio to \( X:1 \) between high-priority threads and medium-priority thread.

- It also sets the processor-time ratios to \( Y:1 \) between medium-priority threads and low-priority threads. \( X \) and \( Y \) are integers.

- Example:

\[
\{\text{Property.put priorities p(high:10 medium:10)}\}
\]

- Now let us make our producer-consumer program work. We give the producer low priority, and the consumer high. We also set the priority ratios to \( 10:1 \) and \( 10:1 \).
The program with priorities

```plaintext
local L in
(Property, put priorities p(high:10 medium:10))
thread
(Thread, setThisPriority low)
    L = (Generate 0 100000000)
end
thread
(Thread, setThisPriority high)
    (Sum 0 L)
end
```

Active Objects

Building active objects with ports

- Here is a simple active object:
  ```plaintext
declare P in
local Xs in
    {NewPort Xs P}
thread
    {ForAll Xs proc {$ X} {Browse X} end} end
end
```

Active objects with classes

- An active object’s behavior can be defined by a class
- The class is used to create a (passive) object, which is invoked by one thread that reads from a port’s stream
- Anyone can send a message to the object asynchronously, and the object will execute them one after the other, in sequential fashion:
  ```plaintext
declare ActObj in
local Obj Xs P in
    Obj={New Class init}
    {NewPort Xs P}
thread
    {ForAll Xs proc {$ M} {Obj M} end}
end
proc
    {$ M} {Send P M}
end
end
```

Creating active objects with NewActive

- We can create a function NewActive that behaves like New except that it creates an active object:
  ```plaintext
  fun {NewActive Class Init}
  in
  Obj Xs P
  (Obj={New Class Init})
  (NewPort Xs P)
  thread [ForAll Xs proc {$ M} {Proc X} end]
  proc [M] {Send P M} end
  end
  ActObj = {NewActive Class init}
  end
  ```

Making active objects synchronous

- We can make an active object synchronous by using a dataflow variable to store a result, and waiting for the result before continuing
  ```plaintext
  fun {NewSynchronousActive Class Init}
  in
  Obj Xs P
  (Obj={New Class Init})
  (NewPort Xs P)
  thread [ForAll Xs proc {$ M} {Proc M} end]
  proc [M] {Send P M} end
  end
  ActObj = {NewActive Class init}
  ```

- This can be modified to handle when the active object raises an exception, to pass the exception back to the caller
Playing catch

```plaintext
class Bounce
  attr
    other: Other
    count: Integer
  meth
    init(other: Other)
      other := Other
    end
    ball
      count := count + 1
      other ball
    end
    get(X)
      X := count
    end
  end

declare
  B1, B2
in
  B1 = {NewActive Bounce init(B2)}
  B2 = {NewActive Bounce init(B1)}
end

% Get the ball bouncing
{B1 ball}
% Follow the bounces
{Browse {B1 get($)}}
```

An area server

```plaintext
class AreaServer
  attr
  handlers: [Handler]
  meth
    init
    handlers := nil
  end
    square(X: Integer)
      A := X * X
    end
    circle(R: Integer)
      A := 3.14 * R * R
    end
  end

declare
  S
in
  S = {NewActive AreaServer init}
end

% Query the server
{S square(10 A)}
{Browse A}
{S circle(20 A)}
{Browse A}
```

Event manager with active objects

- An event manager contains a set of event handlers.
- Each handler is a tuple \([I, F, S]\) where \(I\) identifies it, \(F\) is the state update function, and \(S\) is the state.
- Reception of an event causes all triples to be replaced by \([I, F, \{E, S\}]\).
  (transition from \(F\) to \(\{E, S\}\)).
- The manager \(EM\) is an active object with four methods:
  - \{EM init\} initializes the event manager.
  - \{EM event(E)\} posts event \(E\) at the manager.
  - \{EM add(F, S, I)\} adds new handler with \(F\), \(S\), and returns \(I\).
  - \{EM delete(I, S)\} removes handler \(I\), returns state.
- This example taken from real use in Erlang.

Defining the event manager

- Mix of functional and object-oriented style.

```plaintext
class EventManager
  attr
    handlers: [Handler]
  meth
    init
      handlers := nil
    end
    event(E)
      handlers :=
          Map @handlers
          fun {Id, F, S} \{Id, F, \{E, S\}\} end
    end
    add(F, S, I)
      Id := {NewName}
      handlers := Id # F # S | @handlers
    end
    delete(I, S)
      handlers := List.partition
          @handlers
          fun {Id, F, S} I == Id end
        \{Id, F, S\} end
    end
  end
```

Using the event manager

- Simple memory-based handler keeps list of events.

```plaintext
declare EM MemH Id in
  EM := {NewActive EventManager init}
MemH = fun {E, Buf} E | Buf end
{EM add(MemH nil Id)}
{EM event(a1)}
{EM event(a2)}
... 
```

- An event handler is purely functional, yet when put in the event manager, the latter is a concurrent imperative program. This is an example of separation of concerns by using multiple paradigms.
Peer-to-peer systems (1)

- Network transparency works well for a small number of nodes; what do we do when the number of nodes becomes very large?
  - This is what is happening now
- We need a scalable way to handle large numbers of nodes
  - Peer-to-peer systems provide one solution
  - A distributed system that connects resources located at the edges of the Internet
  - Resources: storage, computation power, information, etc.
  - Peer software: all nodes are functionally equivalent
- Dynamic
  - Peers join and leave frequently
  - Failures are unavoidable

Peer-to-peer systems (2)

- Unstructured systems
  - Napster (first generation): still had centralized directory
  - Gnutella, Kazaa, ... (second generation): neighbor graph, fully decentralized but no guarantees, often uses superpeer structure
- Structured overlay networks (third generation)
  - Using non-random topologies
  - Strong guarantees on routing and message delivery
  - Testing on realistically harsh environments (e.g., PlanetLab)
  - DHT (Distributed Hash Table): provides lookup functionality
  - Many examples: Chord, CAN, Tapestry, P-Grid, DKS, Viceroy, Tango, Koorde, etc.

Examples of P2P networks

- Hybrid (client/server)
  - Napster
- Unstructured P2P
  - Gnutella
- Structured P2P
  - Exponential network
    - DHT (Distributed Hash Table), e.g., Chord

Properties of structured overlay networks

- Scalable
  - Works for any number of nodes
- Self-organizing
  - Routing tables updated with node joins/leaves
  - Routing tables updated with node failures
- Provides guarantees
  - If operated inside of failure model, then communication is guaranteed within an upper bound on number of hops
  - Broadcast can be done with a minimum number of messages
- Provides basic services
  - Name-based communication (point-to-point and group)
  - DHT (Distributed Hash Table): efficient storage and retrieval of (key,value) pairs

Self organization

- Maintaining the routing tables
  - Correction-on-use (lazy approach)
  - Periodic correction (eager approach)
  - Guided by assumptions on traffic
- Cost
  - Depends on structure
  - A typical algorithm, DKS (distributed k-ary search), achieves logarithmic cost for reconfiguration and for key resolution (lookup)
- Example of lookup for Chord, the first well-known structured overlay network

Chord: lookup illustrated

Given a key, find the value associated to the key
(here, the value is the IP address of the node that stores the key)

Assume node # searches for the value associated to key K with virtual identifier z

<table>
<thead>
<tr>
<th>Interval</th>
<th>node to be contacted</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1)</td>
<td>0</td>
</tr>
<tr>
<td>(1,2)</td>
<td>6</td>
</tr>
<tr>
<td>(2,4)</td>
<td>6</td>
</tr>
<tr>
<td>(4,8)</td>
<td>6</td>
</tr>
<tr>
<td>(8,0)</td>
<td>12</td>
</tr>
</tbody>
</table>

Indicates presence of a node
Self management

The need

- Because of the growth of the Internet, individual computers are so numerous that they tend to vanish – Programs that run on a single computer will become the exception
- Programs will be spread over many computers
- This brings enormous complexity problems
- Network transparency solves a few: application code remains simple despite the distribution
- Structured overlay networks solve a few more: self-organizing communications and storage infrastructure
- But the many services and applications, all built on top, introduce much more complexity
- A solution: self management

- We need a general architecture for building self-managing systems
- A key part of this architecture is a powerful component model

Self management

- The system should be able to reconfigure itself to handle changes in its environment or its requirements without human intervention but according to high-level management policies
- Human intervention is lifted to the level of the policies
- Typical self-management operations include: add/remove nodes, tune performance, auto-configure, failure detection & recovery, intrusion detection & recovery, software rejuvenation
- Self management exists at all levels
  - Such as: application level, service level, cluster level, process/OS level
- For large-scale systems, environmental changes that require recovery by the system become normal and even frequent events
  - "Abnormal" events are normal occurrences (e.g., failure is a normal occurrence)

Axes of self management

- Self configuration: adding or removing nodes during execution, version updating during execution, lifecycle maintenance
- Self healing ("fault tolerance"): according to failure model (node failure, network failure)
- Self tuning: load balancing and overload management
- Self protection ("security"): according to threat model (simple model: nodes themselves are trustworthy)

Different layers, different requirements

- Application level
  - Software design, component software
  - "Autonomic Computing" techniques: removing human from the loop
- Services
  - Load-balancing for service infrastructure
  - Matrix of service infrastructure
  - Failure and recovery of services
  - Data management and replication
- Cluster level
  - Resource management and replication
  - Fault-tolerance based cluster infrastructure
- Cluster level
  - Load-balancing for service infrastructure
  - Self-management services (e.g., resource control)
  - Load-balancing services
- Node replication and replacement
- Process/OS level
  - Node protection mechanisms (e.g., intrusion detection)
  - Software rejuvenation
  - Failure and recovery infrastructure

Mechanisms for self management

- Self management adds feedback loops throughout the system
- Detection of anomaly → calculation of a correction → application of the correction
- In non-self-managing systems, the feedback loops often go through human beings
- In self-managing systems, the human is no longer inside the loop but manages the loop from the outside
- The human manages the policy, the system implements the mechanism
- Problems of global behavior: convergence, oscillation, chaos, divergence
  - Techniques from cybernetics and general system theory
  - Software systems are usually non-linear and non-monotonic: Is a linear or monotonic approximation possible? Can the system be forced into a linear or monotonic mode?
Simple feedback loop

- Self-management through feedback loops
- Program modules must detect (introspection) and actuate (control)
- Whole modules might be devoted to being detectors or actuators
  - Ex. synchrony, failure detection, not do much, implemented for calculation (biological detection, belief propagation)

Complexity of interacting feedback loops

- Feedback loops exist throughout the system
- Problems of global behavior
  - Does it converge or diverge?
  - Does it oscillate or behave chaotically?
- Analysis not always easy
  - Linear and monotonic loops are easy; unfortunately software is usually nonlinear
  - Interaction between nested feedback loops (e.g., Norbert Wiener’s classic example of a fire in an airconditioned hotel)
- What are the rules of good feedback design?
  - Analogous to structured and object-oriented programming
  - We need to understand how to build general self-managing systems