Concurrent Programming in Oz (VRH Chs 1,4,5,8)

Carlos Varela
RPI
March 12, 2007

Adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

Concurrency

- Some programs are best written as a set of activities that run independently (concurrent programs)
- Concurrency is essential for interaction with the external environment
- Examples includes GUI (Graphical User Interfaces), operating systems, web services
- Also programs that are written independently but interact only when needed (client-server, peer-to-peer applications)
- This lecture is about declarative concurrency, programs with no observable nondeterminism, the result is a function
- Independent procedures that execute on their pace and may communicate through shared dataflow variables

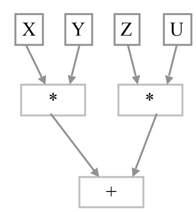
Concurrency

- How to do several things at once
- Concurrency: running several activities each running at its own pace
- A *thread* is an executing sequential program
- A program can have multiple threads by using the thread instruction
- {Browse 99*99} can immediately respond while Pascal is computing

```
thread
  P in
  P = {Pascal 21}
  {Browse P}
end
{Browse 99*99}
```


Dataflow

- What happens when multiple threads try to communicate?
- A simple way is to make communicating threads synchronize on the availability of data (data-driven execution)
- If an operation tries to use a variable that is not yet bound it will wait
- The variable is called a *dataflow variable*



Dataflow (II)

- Two important properties of dataflow
  - Calculations work correctly independent of how they are partitioned between threads (concurrent activities)
  - Calculations are patient, they do not signal error; they wait for data availability
- The dataflow property of variables makes sense when programs are composed of multiple threads

```
declare X
thread
  {Delay 5000} X=99
End
{Browse 'Start'} {Browse X*X}
```

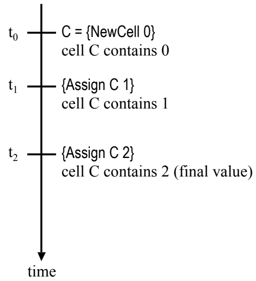
```
declare X
thread
  {Browse 'Start'} {Browse X*X}
end
{Delay 5000} X=99
```


Nondeterminism

- What happens if a program has both concurrency and state together?
- This is very tricky
- The same program can give different results from one execution to the next
- This variability is called *nondeterminism*
- Internal nondeterminism is not a problem if it is not observable from outside

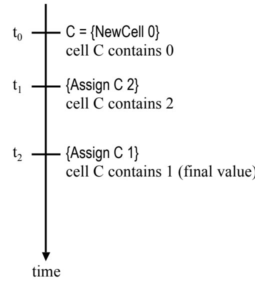
Nondeterminism (2)

```
declare
C = {NewCell 0}

thread {Assign C 1} end
thread {Assign C 2} end
```



Nondeterminism (3)

```
declare
C = {NewCell 0}

thread {Assign C 1} end
thread {Assign C 2} end
```



Nondeterminism (4)

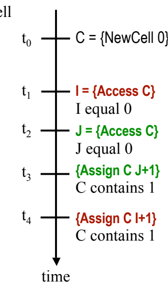
```
declare
C = {NewCell 0}

thread I in
   I = {Access C}
   {Assign C I+1}
end
thread J in
   J = {Access C}
   {Assign C J+1}
end
```

- What are the possible results?
- Both threads increment the cell C by 1
- Expected final result of C is 2
- Is that all?

Nondeterminism (5)

- Another possible final result is the cell C containing the value 1

```
declare
C = {NewCell 0}
thread I in
   I = {Access C}
   {Assign C I+1}
end
thread J in
   J = {Access C}
   {Assign C J+1}
end
```



Lessons learned

- Combining concurrency and state is tricky
- Complex programs have many possible *interleavings*
- Programming is a question of mastering the interleavings
- Famous bugs in the history of computer technology are due to designers overlooking an interleaving (e.g., the Therac-25 radiation therapy machine giving doses 1000's of times too high, resulting in death or injury)
- If possible try to avoid concurrency and state together
- Encapsulate state and communicate between threads using dataflow
- Try to master interleavings by using *atomic operations*

Atomicity

- How can we master the interleavings?
- One idea is to reduce the number of interleavings by programming with coarse-grained atomic operations
- An operation is *atomic* if it is performed as a whole or nothing
- No intermediate (partial) results can be observed by any other concurrent activity
- In simple cases we can use a *lock* to ensure atomicity of a sequence of operations
- For this we need a new entity (a lock)

Atomicity (2)

```
declare
L = {NewLock}

lock L then                     lock L then
  sequence of ops 1  } Thread 1    sequence of ops 2  } Thread 2
end                             end
```


The program

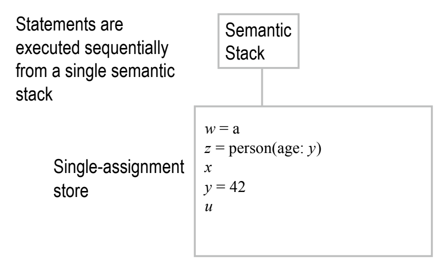
```
declare
C = {NewCell 0}
L = {NewLock}

thread
   lock L then I in
      I = {Access C}
      {Assign C I+1}
   end
end
thread
   lock L then J in
      J = {Access C}
      {Assign C J+1}
   end
end
```

The final result of C is always 2

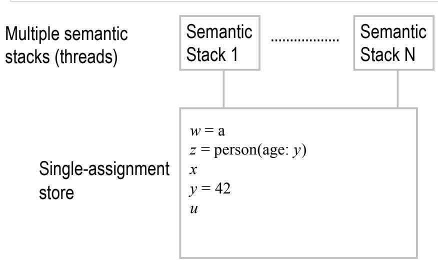
Review of concurrent programming

- There are four basic approaches:
  - Sequential programming (no concurrency)
  - Declarative concurrency (streams in a functional language, Oz)
  - Message passing with active objects (Erlang, SALSA)
  - Atomic actions on shared state (Java)
- The atomic action approach is the *most difficult*, yet it is the one you will probably be most exposed to!
- But, if you have the choice, which approach to use?
  - Use the simplest approach that does the job: sequential if that is ok, else declarative concurrency if there is no observable nondeterminism, else message passing if you can get away with it.

The sequential model



The concurrent model

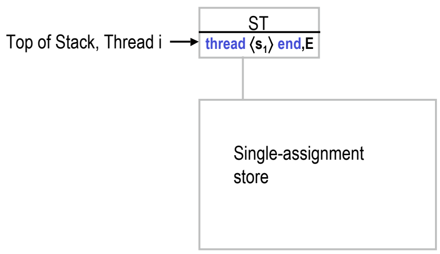


Concurrent declarative model

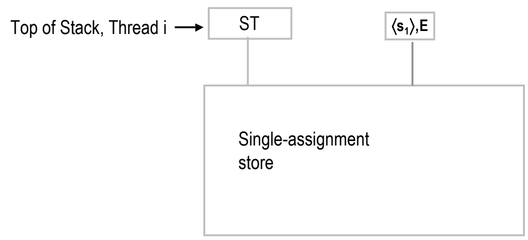
The following defines the syntax of a statement, ⟨s⟩ denotes a statement

```
⟨s⟩ ::= skip                                          empty statement
     |  ⟨x⟩ = ⟨y⟩                                     variable-variable binding
     |  ⟨x⟩ = ⟨v⟩                                     variable-value binding
     |  ⟨s1⟩ ⟨s2⟩                                     sequential composition
     |  local ⟨x⟩ in ⟨s1⟩ end                          declaration
     |  proc {⟨x⟩ ⟨y1⟩ … ⟨yn⟩ } ⟨s1⟩ end               procedure introduction
     |  if ⟨x⟩ then ⟨s1⟩ else ⟨s2⟩ end                 conditional
     |  { ⟨x⟩ ⟨y1⟩ … ⟨yn⟩ }                             procedure application
     |  case ⟨x⟩ of ⟨pattern⟩ then ⟨s1⟩ else ⟨s2⟩ end  pattern matching
     |  thread ⟨s1⟩ end                                thread creation
```

The concurrent model



The concurrent model



Basic concepts

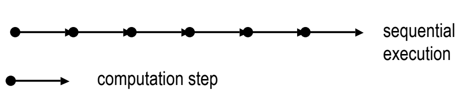
- The model allows multiple statements to execute ”at the same time”.
- Imagine that these threads really execute in parallel, each has its own processor, but share the same memory
- Reading and writing different variables can be done simultaneously by different threads, as well as reading the same variable
- Writing the same variable is done sequentially
- The above view is in fact equivalent to an *interleaving execution*: a totally ordered sequence of computation steps, where threads take turns doing one or more steps in sequence

Causal order

- In a sequential program all execution states are totally ordered
- In a concurrent program all execution states of a given thread are totally ordered
- The execution state of the concurrent program as a whole is partially ordered

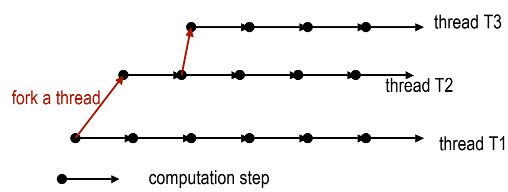
Total order

- In a sequential program all execution states are totally ordered

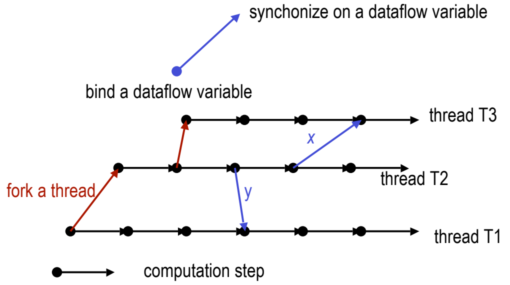


Causal order in the declarative model

- In a concurrent program all execution states of a given thread are totally ordered
- The execution state of the concurrent program is partially ordered



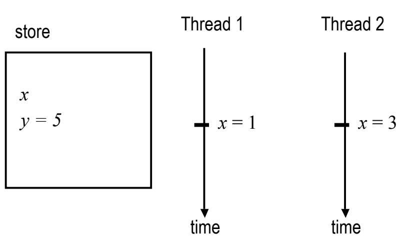
Causal order in the declarative model



Nondeterminism

- An execution is nondeterministic if there is a computation step in which there is a choice what to do next
- Nondeterminism appears naturally when there is concurrent access to shared state

Example of nondeterminism



The thread that binds x first will continue, the other thread will raise an exception

Nondeterminism

- An execution is nondeterministic if there is a computation step in which there is a choice what to do next
- Nondeterminism appears naturally when there is concurrent access to shared state
- In the concurrent declarative model when there is only one binder for each dataflow variable, the nondeterminism is not observable on the store (i.e. the store develops to the same final results)
- This means for correctness we can ignore the concurrency

Scheduling

- The choice of which thread to execute next and for how long is done by a part of the system called the *scheduler*
- A thread is *runnable* if its next statement to execute is not blocked on a dataflow variable, otherwise the thread is *suspended*
- A scheduler is fair if it does not starve a runnable thread, i.e. all runnable threads eventually execute
- Fair scheduling makes it easy to reason about programs and program composition
- Otherwise some correct program (in isolation) may never get processing time when composed with other programs

The semantics

- In the sequential model we had:

  (ST, σ)

ST is a stack of semantic statements

σ is the single assignment store

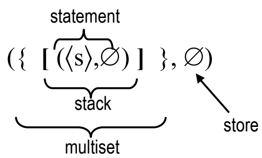
- In the concurrent model we have:

  (MST, σ)

MST is a (multi)set of stacks of semantic statements

σ is the single assignment store

The initial execution state



Execution (the scheduler)

- At each step, one runnable semantic stack is *selected* from MST (the multiset of stacks), call it ST, s.t.
  MST = ST ∪ MST'
- Assume the current store is σ, one computation step is done that transforms ST to ST' and σ to σ'
- The total computation state is transformed from (MST, σ) to (ST' ∪ MST', σ')
- Which stack is selected, and how many steps are taken is the task of the scheduler, a good scheduler should be fair, i.e., each runnable 'thread' will eventually be selected
- The computation stops when there are no runnable stacks

Example of runnable threads

```
proc {Loop P N}
  if N > 0 then
    {P} {Loop P N-1}
  else skip end
end
thread {Loop
        proc {$} {Show 1} end
        1000}
end
thread {Loop
        proc {$} {Show 2} end
        1000}
end
```

- This program will interleave the execution of two threads, one printing 1, and the other printing 2
- We assume a fair scheduler

Dataflow computation

- Threads suspend on data unavailability in dataflow variables
- The **{Delay X}** primitive makes the thread suspends for X milliseconds, after that, the thread is runnable

```
declare X
{Browse X}
local Y in
  thread {Delay 1000} Y = 10*10 end
  X = Y + 100*100
end
```


Illustrating dataflow computation

```
declare X0 X1 X2 X3
{Browse [X0 X1 X2 X3]}
thread
  Y0 Y1 Y2 Y3
in
  {Browse [Y0 Y1 Y2 Y3]}
  Y0 = X0 + 1
  Y1 = X1 + Y0
  Y2 = X2 + Y1
  Y3 = X3 + Y2
  {Browse completed}
end
```

- Enter incrementally the values of X0 to X3
- When X0 is bound the thread will compute Y0=X0+1, and will suspend again until X1 is bound

Concurrent Map

```
fun {Map Xs F}
  case Xs
  of nil then nil
  [] X|Xr then
    thread {F X} end|{Map Xr F}
  end
end
```

- This will fork a thread for each individual element in the input list
- Each thread will run only if both the element X and the procedure F is known

Concurrent Map Function

```
fun {Map Xs F}
      case Xs
      of nil then nil
      [] X|Xr then thread {F X} end |{Map Xr F}
      end
end
```

- What this looks like in the kernel language:

```
proc {Map Xs F Rs}
    case Xs
    of nil then Rs = nil
    [] X|Xr then R Rr in
       Rs = R|Rr
       thread R = {F X} end
       {Map Xr F Rr}
    end
end
```


How does it work?

- If we enter the following statements:
  declare F X Y Z
  {Browse **thread** {Map X F} **end**}
- A thread executing Map is created.
- It will suspend immediately in the case-statement because X is unbound.
- If we thereafter enter the following statements:
  X = 1|2|Y
  fun {F X} X*X **end**
- The main thread will traverse the list creating two threads for the first two arguments of the list

How does it work?

- The main thread will traverse the list creating two threads for the first two arguments of the list:
  thread {F 1} **end**, and **thread** {F 2} **end**,

After entering:

Y = 3|Z
Z = nil

the program will complete the computation of the main thread and the newly created thread **thread** {F 3} **end**, resulting in the final list [1 4 9].

Simple concurrency with dataflow

- Declarative programs can be easily made concurrent
- Just use the thread statement where concurrency is needed

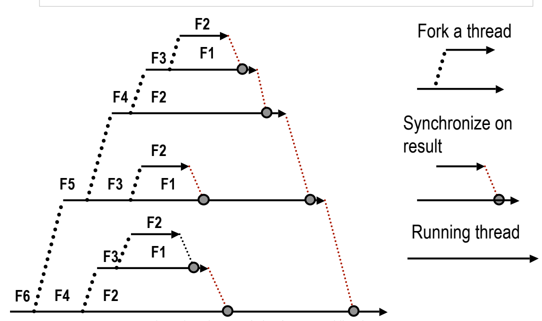
```
fun {Fib X}
  if X=<2 then 1
  else
    thread {Fib X-1} end + {Fib X-2}
  end
end
```


Understanding why

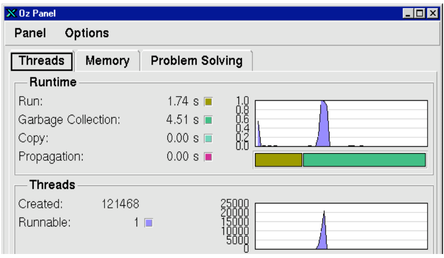
```
fun {Fib X}
  if X=<2 then 1
  else F1 F2 in
    F1 = thread {Fib X-1} end
    F2 = {Fib X-2}

    F1 + F2
  end
end
```

Dataflow dependency

Execution of {Fib 6}



Threads and Garbage Collection



Concurrency and state are tough when used together

- Execution consists of multiple threads, all executing independently and all using shared memory
- Because of interleaving semantics, execution happens as if there was one global order of operations
- Assume two threads and each thread does k operations. Then the total number of possible interleavings is $\binom{2k}{k}$. This is exponential in k.
- One can program by reasoning on all possible interleavings, but this is extremely hard. What do we do?

Concurrent stateful model

⟨s⟩	::=	skip	*empty statement*
	|	⟨x⟩ = ⟨y⟩	*variable-variable binding*
	|	⟨x⟩ = ⟨v⟩	*variable-value binding*
	|	⟨s_1⟩ ⟨s_2⟩	*sequential composition*
	|	local ⟨x⟩ in ⟨s_1⟩ end	*declaration*
	|	proc { ⟨x⟩ ⟨y_1⟩ … ⟨y_n⟩ } ⟨s_1⟩ end	*procedure creation*
	|	if ⟨x⟩ then ⟨s_1⟩ else ⟨s_2⟩ end	*conditional*
	|	{ ⟨x⟩ ⟨y_1⟩ … ⟨y_n⟩ }	*procedure application*
	|	case ⟨x⟩ of ⟨pattern⟩ then ⟨s_1⟩ else ⟨s_2⟩ end	*pattern matching*
	|	{NewName ⟨x⟩ }	*name creation*
	|	**thread ⟨s⟩ end**	***thread creation***
	|	{ByNeed ⟨x⟩ ⟨y⟩ }	*trigger creation*
	|	try ⟨s_1⟩ catch ⟨x⟩ then ⟨s_2⟩ end	*exception context*
	|	raise ⟨x⟩ end	*raise exception*
	|	**{NewCell ⟨x⟩ ⟨y⟩ }**	***cell creation***
	|	**{Exchange ⟨x⟩ ⟨y⟩ ⟨z⟩ }**	***cell exchange***

Why not use a simpler model?

- The concurrent declarative model is much simpler
  - Programs give the same results as if they were sequential, but they give the results incrementally
- Why is this model so easy?
  - Because dataflow variables can be bound to only one value. A thread that shares a variable with another thread does not have to worry that the other thread will change the binding.
- So why not stick with this model?
  - In many cases, we can stick with this model
  - But not always. For example, two clients that communicate with one server cannot be programmed in this model. Why not? Because there is an observable nondeterminism.
- The concurrent declarative model is deterministic. If the program we write has an observable nondeterminism, then we cannot use the model.

Programming with concurrency and state

- Programming with concurrency and state is largely a matter of reducing the number of interleavings, so that we can reason about programs in a simpler way. There are two basic approaches: message passing and atomic actions.
- Message passing with active objects: Programs consist of threads that send asynchronous messages to each other. Each thread only receives a message when it is ready, which reduces the number of interleavings.
- Atomic actions on shared state: Programs consist of passive objects that are called by threads. We build large atomic actions (e.g., with locks, monitors, or transactions) to reduce the number of interleavings.

When to use each approach

- Message passing: useful for multi-agent applications, i.e., programs that consist of autonomous entities (« agents », « actors » or « active objects ») that communicate with each other.
- Atomic actions: useful for data-centered applications, i.e., programs that consist of a large repository of data (« database » or « shared state ») that is accessed and updated concurrently.
- Both approaches can be used together in the same application, for different parts

Ports and cells

- We have seen ***cells***, the basic unit of encapsulated state, as a primitive concept underlying stateful and object-oriented programming. Cells are like variables in imperative languages.
- Cells are the natural concept for programming with shared state
- There is another way to add state to a language, which we call a ***port***. A port is an asynchronous FIFO communications channel.
- Ports are a natural concept for programming with active objects
- Cells and ports are *duals* of each other
  - Each can be implemented with the other, so they are equal in expressiveness
  - Each is more natural in some circumstances
  - They are equivalent because each allows many-to-one communication (cell shared by threads, port shared by threads)

Ports

- A port is an ADT with two operations:
  - {NewPort S P}: create a new port P with a new stream S. The stream is a list with unbound tail, used to model the FIFO nature of the communications channel.
  - {Send P X}: send message X on port P. The message is appended to the stream S and can be read by threads reading S.
- Example:

```
declare P S in
   {NewPort S P}
   {Browse S}
   thread{Send P 1}end
   thread{Send P 2}end
```


Building locks with cells

- The basic way to program with shared state is by using locks
- A lock is a region of the program that can only be occupied by one thread at a time. If a second thread attempts to enter, it will suspend until the first thread exits.
- More sophisticated versions of locks are monitors and transactions:
  - Monitors: locks with a gating mechanism (e.g., wait/notify in Java) to control which threads enter and exit and when. Monitors are the standard primitive for concurrent programming in Java.
  - Transactions: locks that have two exits, a normal and abnormal exit. Upon abnormal exit (called « abort »), all operations performed in the lock are undone, as if they were never done. Normal exit is called « commit » .
- Locks can be built with cells. The idea is simple: the cell contains a token. A thread attempting to enter the lock takes the token. A thread that finds no token will wait until the token is put back.

Building active objects with ports

- Here is a simple active object:

```
declare P in
local Xs in
   {NewPort Xs P}
   thread {ForAll Xs proc {$ X} {Browse X} end} end
end

{Send P foo(1)}
thread {Send P bar(2)} end
```


Defining ports with cells

- A port is an unbundled stateful ADT:

```
proc {NewPort S P}
   C={NewCell S}
in
   P={Wrap C}
end

proc {Send P X}
   C={Unwrap P}
   Old
in
   {Exchange C X|Old Old}
end
```

Anyone can do a send because anyone can do an exchange

Exercises

1. VRH Exercise 4.11.3 (page 339)
2. VRH Exercise 4.11.5 (page 339)
3. How would you implement Pi-Calculus processes/channels in Oz?