Concurrent Object-Oriented Programming

Java Concurrency (VRH 8.6)
Objects, Active Objects (VRH 7.2,7.8)

Carlos Varela
RPI

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

October 30, 2006

Concurrent Programming in Java

- Java is multi-threaded.
- Two ways to create new threads:
  - Extend java.lang.Thread
    - Overwrite "run()" method.
  - Implement Runnable interface
    - Include a "run()" method in your class.
- Starting a thread
  - new MyThread().start();
  - new Thread(runnable).start();

The synchronized Statement

- To ensure only one thread can run a block of code, use synchronized:

```
synchronized ( object ) {
   // critical code here
}
```

- Every object contains an internal lock for synchronization.

synchronized as a modifier

- You can also declare a method as synchronized:

```
synchronized int blah(String x) {
 // blah blah blah
}
```

equivalent to:

```
int blah(String x) {
  synchronized (this) {
 // blah blah blah
 }
}
```


Object-Oriented Programming in Oz

The class Counter has the syntactic form

```
class Counter
     attr val
     meth display
          {Browse @val}
     end
     meth inc(Value)
          val := @val + Value
     end
     meth init(Value)
          val := Value
     end
end
```


Attributes of Classes

The class Counter has the syntactic form

```
class Counter
     attr val
     meth display
          {Browse @val}
     end
     meth inc(Value)
          val := @val + Value
     end
     meth init(Value)
          val := Value
     end
end
```

val is an attribute: a modifiable cell that is accessed by the atom val

Attributes of classes

The class Counter has the syntactic form

```
class Counter
    attr val
    meth display
        {Browse @val}
    end
    meth inc(Value)
        val := @val + Value
    end
    meth init(Value)
        val := Value
    end
end
```

the attribute val is accessed by the operator @val

Attributes of classes

The class Counter has the syntactic form

```
class Counter
    attr val
    meth display
        {Browse @val}
    end
    meth inc(Value)
        val := @val + Value
    end
    meth init(Value)
        val := Value
    end
end
```

the attribute val is assigned by the operator := as val := ...

Methods of classes

The class Counter has the syntactic form

```
class Counter
    attr val
    meth display
        {Browse @val}
    end
    meth inc(Value)
        val := @val + Value
    end
    meth init(Value)
        val := Value
    end
end
```

methods are statements method head is a record (tuple) pattern

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
     |  ⟨s1⟩ ⟨s2⟩                                      sequential composition
     |  local ⟨x⟩ in ⟨s1⟩ end                          declaration
     |  proc { ⟨x⟩ ⟨y1⟩ … ⟨yn⟩ } ⟨s1⟩ end              procedure creation
     |  if ⟨x⟩ then ⟨s1⟩ else ⟨s2⟩ end                 conditional
     |  { ⟨x⟩ ⟨y1⟩ … ⟨yn⟩ }                            procedure application
     |  case ⟨x⟩ of ⟨pattern⟩ then ⟨s1⟩ else ⟨s2⟩ end  pattern matching
     |  {NewName ⟨x⟩ }                                 name creation
     |  thread ⟨s⟩ end                                 thread creation
     |  {ByNeed ⟨x⟩ ⟨y⟩ }                              trigger creation
     |  try ⟨s1⟩ catch ⟨x⟩ then ⟨s2⟩ 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

Overview of concurrent programming

- There are four basic approaches:
  - Sequential programming (no concurrency)
  - Declarative concurrency (streams in a functional language)
  - 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.

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

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:

```
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 {ActObj M} {Send P M} end
end
{ActObj msg(1)}
```

- Note that {Obj M} is synchronous and {ActObj M} is asynchronous!

Creating active objects with NewActive

- We can create a function NewActive that behaves like New except that it creates an active object:

```
fun {NewActive Class Init}
    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

ActObj = {NewActive Class init}
```

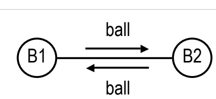

Making active objects synchronous

- We can make an active object synchronous by using a dataflow variable to store a result, and waiting for the result before continuing

```
fun {NewSynchronousActive Class Init}
    Obj Xs P
in
    Obj={New Class Init}
    {NewPort Xs P}
    thread {ForAll Xs proc {$ msg(M X)} {Obj M} X=unit end} end
    proc {$ M} X in {Send P msg(M X)} {Wait X} end
end
```

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

Playing catch




```
class Bounce
    attr other count:0
    meth init(Other)
        other:=Other
    end
    meth ball
        count:=@count+1
        {@other ball}
    end
    meth get(X)
        X=@count
    end
end
```

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

% Get the ball bouncing
{B1 ball}

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

An area server

```
class AreaServer
    meth init skip end
    meth square(X A)
        A=X*X
    end
    meth circle(R A)
        A=3.14*R*R
    end
end
```

```
declare S in
S={NewActive AreaServer init}

% Query the server
declare A in
{S square(10 A)}
{Browse A}

declare A in
{S circle(20 A)}
{Browse A}
```


Event manager with active objects

- An event manager contains a set of event handlers
- Each handler is a triple `Id#F#S` where Id identifies it, F is the state update function, and S is the state
- Reception of an event causes all triples to be replaced by `Id#F#{F E S}` *(transition from F to {F 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 Id)}` adds new handler with F, S, and returns Id
  - `{EM delete(Id S)}` removed handler Id, returns state
- This example taken from real use in Erlang

Defining the event manager

- Mix of functional and object-oriented style

```
class EventManager
   attr handlers
   meth init handlers:=nil end
   meth event(E)
      handlers:=
         {Map @handlers fun {$ Id#F#S} Id#F#{F E S} end}
   end
   meth add(F S Id)
      Id={NewName}
      handlers:=Id#F#S|@handlers
   end
   meth delete(DId DS)
      handlers:={List.partition
         @handlers fun {$ Id#F#S} DId==Id end [_#_#DS]}
   end
end
```

State transition done using functional programming

Using the event manager

- Simple memory-based handler keeps list of events

```
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.

Exercises

60. Do Java and C++ provide linguistic abstractions for active objects? If so, which? If not, how would you implement this abstraction?
61. Exercise VRH 7.9.1 (pg 567)
62. *Exercise VRH 7.9.6(a) (pg 568)