Programming Languages
(CSCI 4430/6430)
Part 2: Concurrent Programming: Summary

Carlos Varela
Rensselaer Polytechnic Institute

November 3, 2017
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, otherwise use actors and message passing.
Actors/SALSA

- **Actor Model**
  - A reasoning framework to model concurrent computations
  - Programming abstractions for distributed open systems

- **SALSA**
  - Simple Actor Language System and Architecture
  - An actor-oriented language for mobile and internet computing
  - Programming abstractions for internet-based concurrency, distribution, mobility, and coordination
1. Extend a functional language (λ-calculus + ifs and pairs) with actor primitives.

2. Define an operational semantics for actor configurations.

3. Study various notions of equivalence of actor expressions and configurations.

4. Assume fairness:
   – Guaranteed message delivery.
   – Individual actor progress.
λ-Calculus as a Model for Sequential Computation

Syntax

\[ e ::= \ \text{v} \quad \text{value} \]
\[ \quad | \ \lambda v.e \quad \text{functional abstraction} \]
\[ \quad | ( \ e \ e ) \quad \text{application} \]

Example of beta-reduction:

\[ (\ \lambda x.x^2 \ \ 2 ) \rightarrow x^2\{2/x\} \]
Actor Primitives

• **send**(a, v)
  - Sends value v to actor a.

• **new**(b)
  - Creates a new actor with behavior b (a λ-calculus abstraction) and returns the identity/name of the newly created actor.

• **ready**(b)
  - Becomes ready to receive a new message with behavior b.
AMST Actor Language
Examples

\[ b5 = \text{rec}(\lambda y. \lambda x. \text{seq}(\text{send}(x,5), \text{ready}(y))) \]
receives an actor name \( x \) and sends the number 5 to that actor, then it becomes ready to process new messages with the same behavior \( y \).

Sample usage:
\[ \text{send}(\text{new}(b5), a) \]

A \textit{sink}, an actor that disregards all messages:
\[ \text{sink} = \text{rec}(\lambda b. \lambda m. \text{ready}(b)) \]
cell = rec(\b. \c. \m.
    if ( get?(m),
        seq( send(cust(m), c),
            ready(b(c))),
    if ( set?(m),
        ready(b(contents(m))),
        ready(b(c)))))

Using the cell:
let a = new(cell(0)) in seq( send(a, mkset(7)),
    send(a, mkset(2)),
    send(a, mkget(c)))
Consider:

\[
\text{treeprod} = \text{rec}(\lambda f. \lambda \text{tree}.
\quad \text{if}(\text{isnat}(\text{tree}),
\quad \text{tree},
\quad f(\text{left}(\text{tree})) \times f(\text{right}(\text{tree}))))
\]

which multiplies all leaves of a tree, which are numbers.

You can do the “left” and “right” computations concurrently.
Tree Product Behavior

\[ B_{\text{treeprod}} = \]
\[ \text{rec}(\lambda b. \lambda m. \]
\[ \text{seq}(\text{if}(\text{isnat}(\text{tree}(m))), \]
\[ \text{send}(\text{cust}(m), \text{tree}(m)), \]
\[ \text{let newcust} = \text{new}(B_{\text{joincont}}(\text{cust}(m))), \]
\[ \text{lp} = \text{new}(B_{\text{treeprod}}), \]
\[ \text{rp} = \text{new}(B_{\text{treeprod}}) \text{ in} \]
\[ \text{seq}(\text{send}(\text{lp}, \]
\[ \text{pr(} \text{left}(\text{tree}(m)), \text{newcust})\), \]
\[ \text{send}(\text{rp}, \]
\[ \text{pr(} \text{right}(\text{tree}(m)), \text{newcust})\)), \]
\[ \text{ready}(b)) \]
Tree Product (continued)

\[ B_{\text{joincont}} = \lambda \text{cust}. \lambda \text{firstnum}. \text{ready}(\lambda \text{num}. \text{seq}(\text{send}(\text{cust}, \text{firstnum} \times \text{num}), \text{ready}(\text{sink}))) \]
Operational Semantics for AMST Actor Language

• Operational semantics of actor model as a labeled transition relationship between actor configurations.

• Actor configurations model open system components:
  
  – Set of individually named actors
  
  – Messages “en-route”
Actor Configurations

\[ k = \alpha \parallel \mu \]

\( \alpha \) is a function mapping actor names (represented as free variables) to actor states.

\( \mu \) is a multi-set of messages “en-route.”
Labeled Transition Relation

\[
\begin{align*}
\frac{e \rightarrow \chi e'}{
\alpha, [R \triangleright e \triangleright]_a \parallel \mu \quad \xrightarrow{\text{fun:}a}\quad \alpha, [R \triangleright e' \triangleright]_a \parallel \mu}
\end{align*}
\]

\[
\begin{align*}
\alpha, [R \triangleright \text{new}(b) \triangleright]_a \parallel \mu \quad \xrightarrow{\text{new:}a,a'}\quad \alpha, [R \triangleright a' \triangleright]_a, [\text{ready}(b)]_{a'} \parallel \mu
\end{align*}
\]

\[
\alpha, [R \triangleright \text{send}(a', v) \triangleright]_a \parallel \mu \quad \xrightarrow{\text{snd:}a}\quad \alpha, [R \triangleright \text{nil} \triangleright]_a \parallel \mu \uplus \{\langle a' \leftarrow v \rangle\}
\]

\[
\alpha, [R \triangleright \text{ready}(b) \triangleright]_a \parallel \{\langle a \leftarrow v \rangle\} \uplus \mu \quad \xrightarrow{\text{rcv:}a,v}\quad \alpha, [b(v)]_a \parallel \mu
\]
Semantics example summary

\[ k_0 = [\text{send}(\text{new}(b5), a)]_a \parallel \{\} \]
\[ k_6 = [\text{nil}]_a, [\text{ready}(b5)]_b \parallel \{< a \leq 5 >\} \]

This sequence of (labeled) transitions from \(k_0\) to \(k_6\) is called a computation sequence.
Asynchronous communication

\[
k_0 = [\text{ready(cell(0))}]_a \\
\| \{<a<=s(7)> , <a<=s(2)> , <a<=g(c)> \} \\
\]

Three receive transitions are enabled at \(k_0\).

Multiple enabled transitions can lead to \textit{nondeterministic} behavior.

The set of all computations sequences from \(k_0\) is called the \textit{computation tree} \(\tau(k_0)\).
Nondeterministic behavior (1)

\[ k_0 = [\text{ready(cell(0))}]_a \]
\[ \quad || \{ <a<=s(7)>, <a<=s(2)>, <a<=g(c)> \} \]
\[ k_1 \rightarrow^* [\text{ready(cell(7))}]_a \]
\[ \quad || \{ <a<=s(2)>, <a<=g(c)> \} \]

Customer \( c \) will get 2 or 7.

\[ k_1' \rightarrow^* [\text{ready(cell(2))}]_a \]
\[ \quad || \{ <a<=s(7)>, <a<=g(c)> \} \]

Customer \( c \) will get 0.

\[ k_1'' \rightarrow^* [\text{ready(cell(0))}]_a \]
\[ \quad || \{ <a<=s(7)>, <a<=s(2)>, <c<=0> \} \]
Nondeterministic behavior (2)

$k_0 = \left[ \text{ready(cell(0))} \right]_a || \{<a<=s(7)>, <a<=s(2)>, <a<=g(c)>\}$

Order of three receive transitions determines final state, e.g.:

$$k_0 \xrightarrow{[\text{rcv}:a,g(c)]} k_1 \xrightarrow* [\text{rcv}:a,s(7)] \xrightarrow* [\text{rcv}:a,s(2)] k_3$$

$k_f = \left[ \text{ready(cell(2))} \right]_a || \{<c<=0>\}$

Final cell state is 2.
Nondeterministic behavior (3)

\[ k_0 = [\text{ready(cell(0))}]_a \]
\[ \| \{<a<=s(7)>, <a<=s(2)>, <a<=g(c)>\} \]

Order of three receive transitions determines final state, e.g.:

\[ \text{rcv:}\ a,s(2)] \quad \text{rcv:}\ a,g(c)] \quad \text{rcv:}\ a,s(7)] \]

\[ k_0 \xrightarrow{[\text{rcv:}\ a,s(2)]} k_1 \xrightarrow{*} k_2 \xrightarrow{*} k_3 \]

\[ k_f = [\text{ready(cell(7))}]_a \| \{<c<=2>\} \]

Final cell state is 7.
SALSA support for Actors

- Programmers define behaviors for actors. Actors are instances of behaviors.

- Messages are modeled as potential method invocations. Messages are sent asynchronously.

- State is modeled as encapsulated objects/primitive types.

- Tokens represent future message return values. Continuation primitives are used for coordination.
module cell;

behavior Cell {
    Object content;

    Cell(Object initialContent) {
        content = initialContent;
    }

    Object get() { return content; }

    void set(Object newContent) {
        content = newContent;
    }
}
module cell;

behavior Cell {
  Object content;

  Cell(Object initialContent) {
    content = initialContent;
  }

  Object get() { return content; }

  void set(Object newContent) {
    content = newContent;
  }
}

return asynchronously sets token associated to get message.

Implicit control loop: End of message implies ready to receive next message.
module cell;

behavior CellTester {

    void act( String[] args ) {

        Cell c = new Cell(0);
        c <- set(2);
        c <- set(7);
        token t = c <- get();
        standardOutput <- println( t );
    }

}
module cell;

behavior CellTester {

  void act( String[] args ) {

    Cell c = new Cell(0);
    c <- set(2);
    c <- set(7);
    token t = c <- get();
    standardOutput <- println( t );
  }
}

All message passing is asynchronous.

println message is called partial until token t is produced. Only full messages (with no pending tokens) are delivered to actors.
Erlang support for Actors

- Actors in Erlang are modeled as *processes*. Processes start by executing an arbitrary *function*. Related functions are grouped into *modules*.

- Messages can be any Erlang *terms*, e.g., atoms, tuples (fixed arity), or lists (variable arity). Messages are sent asynchronously.

- State is modeled implicitly with function arguments. Actors explicitly call receive to get a message, and must use tail-recursion to get new messages, i.e., control loop is explicit.
Reference Cell in Erlang

```erlang
-module(cell).
-export([cell/1]).

cell(Content) ->
    receive
        {set, NewContent} -> cell(NewContent);
        {get, Customer}   -> Customer ! Content, cell(Content)
    end.
```

**Encapsulated state** `Content`.

**Message handlers**

**Explicit control loop**: Actions at the end of a message need to include tail-recursive function call. Otherwise actor (process) terminates.
-module(cell).
-export([cell/1]).

cell(Content) ->
   receive
   {set, NewContent} -> cell(NewContent);
   {get, Customer}   -> Customer ! Content,
   cell(Content)
   end.
-module(cellTester).
-export([main/0]).

main() -> C = spawn(cell,cell,[0]),
            C!{set,2},
            C!{set,7},
            C!{get,self()},
            receive
            Value ->
            io:format("~w~n",[Value])
          end.
module(cellTester).
-export([main/0]).

main() -> C = spawn(cell,cell,[0]),
            C!{set,2},
            C!{set,7},
            C!{get,self()},
            receive
                Value ->
                    io:format("~w~n", [Value])
            end.

[0] is a list with the arguments to the module’s function. General form:
spawn(module, function, arguments)

self() is a built-in function (BIF) that returns the process id of the current process.

Function calls take the form:
module:function(args)
module treeprod;

behavior TreeProduct {
    void compute(Tree t, UniversalActor c) {
        if (t.isLeaf()) c <- result(t.value());
        else {
            JoinCont newCust = new JoinCont(c);
            TreeProduct lp = new TreeProduct();
            TreeProduct rp = new TreeProduct();
            lp <- compute(t.left(), newCust);
            rp <- compute(t.right(), newCust);
        }
    }
}
Join Continuation in SALSA

```java
module treeprod;
behavior JoinCont {

    UniversalActor cust;
    int first;
    boolean receivedFirst;

    JoinCont(UniversalActor cust){
        this.cust = cust;
        this.receivedFirst = false;
    }

    void result(int v) {
        if (!receivedFirst){
            first = v; receivedFirst = true;
        }
        else // receiving second value
            cust <- result(first*v);
    }
}
```

C. Varela
-module(treeprod).
-export([treeprod/0, join/1]).

treeprod() ->
receive
   {{Left, Right}, Customer} ->
      NewCust = spawn(treeprod, join, [Customer]),
      LP = spawn(treeprod, treeprod, []),
      RP = spawn(treeprod, treeprod, []),
      LP!{Left, NewCust},
      RP!{Right, NewCust};
   {Number, Customer} ->
      Customer ! Number
end,
  treeprod().

join(Customer) -> receive V1 -> receive V2 -> Customer ! V1*V2 end end.
Tree Product Sample Execution

2> TP = spawn(treeprod,treeprod,[]).
<0.40.0>
3> TP ! {{{{5,6},2},{3,4}},self()}.
{{{{5,6},2},{3,4}},<0.33.0>}
4> flush().
Shell got 720
ok
5>
Actor Languages Summary

- Actors are concurrent entities that react to messages.
  - State is completely encapsulated. There is no shared memory!
  - Message passing is asynchronous.
  - Actors can create new actors. Run-time has to ensure fairness.
- AMST extends the call by value lambda calculus with actor primitives. State is modeled as function arguments. Actors use `ready` to receive new messages.
- SALSA extends an object-oriented programming language (Java) with universal actors. State is explicit, encapsulated in instance variables. Control loop is implicit: ending a message handler, signals readiness to receive a new message. Actors are garbage-collected.
- Erlang extends a functional programming language core with processes that run arbitrary functions. State is implicit in the function’s arguments. Control loop is explicit: actors use `receive` to get a message, and tail-form recursive call to continue. Ending a function denotes process (actor) termination.
Concurrency Control in SALSA

- SALSA provides three main coordination constructs:
  - **Token-passing continuations**
    - To synchronize concurrent activities
    - To notify completion of message processing
    - Named tokens enable arbitrary synchronization (data-flow)
  - **Join blocks**
    - Used for barrier synchronization for multiple concurrent activities
    - To obtain results from otherwise independent concurrent processes
  - **First-class continuations**
    - To delegate producing a result to a third-party actor
Token Passing Continuations

- Ensures that each message in the continuation expression is sent after the previous message has been **processed**. It also enables the use of a message handler return value as an argument for a later message (through the `token` keyword).

  - Example:

    ```
    a1 <- m1() @
    a2 <- m2( token );
    ```

    *Send* \(m_1\) to \(a_1\) asking \(a_1\) to forward the result of processing \(m_1\) to \(a_2\) (as the argument of message \(m_2\)).
Token Passing Continuations

- @ syntax using token as an argument is syntactic sugar.
  - Example 1:
    ```
    a1 <- m1() @
    a2 <- m2( token );
    ```
    is syntactic sugar for:
    ```
    token t = a1 <- m1();
    a2 <- m2( t );
    ```
  - Example 2:
    ```
    a1 <- m1() @
    a2 <- m2();
    ```
    is syntactic sugar for:
    ```
    token t = a1 <- m1();
    a2 <- m2():waitFor( t );
    ```
Named Tokens

- Tokens can be named to enable more loosely-coupled synchronization

  - Example:

    ```
    token t1 = a1 <- m1();
    token t2 = a2 <- m2();
    token t3 = a3 <- m3(t1);
    token t4 = a4 <- m4(t2);
    a <- m(t1,t2,t3,t4);
    ```

    *Sending m(...) to a will be delayed until messages m1()..m4() have been processed. m1() can proceed concurrently with m2().*
Join Blocks

• Provide a mechanism for synchronizing the processing of a set of messages.
• Set of results is sent along as a *token* containing an array of results.
  – Example:

```java
UniversalActor[] actors = { searcher0, searcher1, searcher2, searcher3 };

join {
    for (int i=0; i < actors.length; i++){
        actors[i] <- find( phrase );
    }
} @ resultActor <- output( token );
```

*Send the `find( phrase )` message to each actor in actors[] then after all have completed send the result to resultActor as the argument of an output( ... ) message.*
First Class Continuations

• Enable actors to delegate computation to a third party independently of the processing context.

• For example:

```c
int m(...) {
    b <- n(...) @ currentContinuation;
}
```

*Ask (delegate) actor b to respond to this message m on behalf of current actor (self) by processing b’s message n.*
Delegate Example

module fibonacci;

behavior Calculator {

  int fib(int n) {
    Fibonacci f = new Fibonacci(n);
    f <- compute() @ currentContinuation;
  }

  int add(int n1, int n2) {return n1+n2;}

  void act(String args[]) {
    fib(15) @ standardOutput <- println(token);
    fib(5) @ add(token,3) @
    standardOutput <- println(token);
  }

}

fib(15)

is syntactic sugar for:

self <- fib(15)
module fibonacci;

behavior Fibonacci {
    int n;

    Fibonacci(int n) { this.n = n; }

    int add(int x, int y) { return x + y; }

    int compute() {
        if (n == 0) return 0;
        else if (n <= 2) return 1;
        else {
            Fibonacci fib1 = new Fibonacci(n-1);
            Fibonacci fib2 = new Fibonacci(n-2);
            token x = fib1<-compute();
            token y = fib2<-compute();
            add(x,y) @ currentContinuation;
        }
    }
}

void act(String args[]) {
    n = Integer.parseInt(args[0]);
    compute() @ standardOutput<-println(token);
}

C. Varela
module fibonacci2;

behavior Fibonacci {

    int add(int x, int y) { return x + y; }

    int compute(int n) {
        if (n == 0) return 0;
        else if (n <= 2) return 1;
        else {
            Fibonacci fib = new Fibonacci();
            token x = fib <- compute(n-1);
            compute(n-2) @ add(x,token) @ currentContinuation;
        }
    }

    void act(String args[]) {
        int n = Integer.parseInt(args[0]);
        compute(n) @ standardOutput<-println(token);
    }

}
Execution of
salsa Fibonacci 6

Create new actor

Synchronize on result

Non-blocked actor
module treeprod;
import tree.Tree;

behavior JoinTreeProduct {

    int multiply(Object[] results) {
        return (Integer) results[0] * (Integer) results[1];
    }

    int compute(Tree t) {
        if (t.isLeaf()) return t.value();
        else {
            JoinTreeProduct lp = new JoinTreeProduct();
            JoinTreeProduct rp = new JoinTreeProduct();
            join {
                lp <- compute(t.left());
                rp <- compute(t.right());
            } @ multiply(token) @ currentContinuation;
        }
    }
}

Notice we use token-passing continuations (@,token), a join block (join), and a first-class continuation (currentContinuation).
Concurrency control in Erlang

- Erlang uses a _selective receive_ mechanism to help coordinate concurrent activities:
  - **Message patterns and guards**
    - To select the next message (from possibly many) to execute.
    - To receive messages from a specific process (actor).
    - To receive messages of a specific kind (pattern).
  - **Timeouts**
    - To enable default activities to fire in the absence of messages (following certain patterns).
    - To create timers.
  - **Zero timeouts** (after 0)
    - To implement priority messages, to flush a mailbox.
Selective Receive

receive
  MessagePattern1 [when Guard1] ->
    Actions1 ;
  MessagePattern2 [when Guard2] ->
    Actions2 ;
  ...
end

receive suspends until a message in the actor’s mailbox matches any of the patterns including optional guards.

• Patterns are tried in order. On a match, the message is removed from the mailbox and the corresponding pattern’s actions are executed.

• When a message does not match any of the patterns, it is left in the mailbox for future receive actions.

C. Varela
Selective Receive Example

Example program and mailbox (head at top):

```
receive
    msg_b -> ...
end

receive  tries to match msg_a  and fails.  msg_b can be
         matched, so it is processed.  Suppose execution continues:
```

```
receive
    msg_c -> ...
    msg_a -> ...
end

The next message to be processed is msg_a  since it is the
next in the mailbox and it matches the 2\textsuperscript{nd} pattern.
```
Receiving from a specific actor

Actor ! {self(), message}

self() is a Built-In-Function (BIF) that returns the current (executing) process id (actor name). Ids can be part of a message.

receive
  {ActorName, Msg} when ActorName == A1 ->
  ...
end

receive can then select only messages that come from a specific actor, in this example, A1. (Or other actors that know A1’s actor name.)
counter(Val) ->
  receive
    increment -> counter(Val+1);
    {From,value} ->
      From ! {self(), Val},
      counter(Val);
    stop -> true;
    Other -> counter(Val)
  end.

**counter** is **a behavior that can receive** **increment** **messages**, **value request messages**, and **stop messages**. **Other** **message kinds are ignored**.
Order of message patterns matters

```
receive
    {{Left, Right}, Customer} ->
        NewCust = spawn(treeprod,join,[Customer]),
        LP = spawn(treeprod,treeprod,[]),
        RP = spawn(treeprod,treeprod,[]),
        LP!{Left,NewCust},
        RP!{Right,NewCust};
    {Number, Customer} ->
        Customer ! Number
end
```

In this example, a binary tree is represented as a tuple

\[
{\{\{5,6\},2\},\{3,4\}}
\]

\{Left,Right\} is a more specific pattern than Number is (which matches anything!). Order of patterns is important.
Selective Receive with Timeout

receive
  MessagePattern1 [when Guard1] ->
    Actions1 ;
  MessagePattern2 [when Guard2] ->
    Actions2 ;
...
  after TimeOutExpr ->
    ActionsT
end

TimeOutExpr evaluates to an integer interpreted as milliseconds.

If no message has been selected within this time, the timeout occurs and ActionsT are scheduled for evaluation.

A timeout of infinity means to wait indefinitely.
sleep(Time) ->
    receive
        after Time ->
            true
    end.

sleep(Time) suspends the current actor for Time milliseconds.
Timeout Example

receive
  click ->
    receive
      click ->
        double_click
        after double_click_interval() ->
          single_click
    end
  end
...
end

double_click_interval evaluates to the number of milliseconds expected between two consecutive mouse clicks, for the receive to return a double_click. Otherwise, a single_click is returned.
Zero Timeout

receive
    MessagePattern1 [when Guard1] ->
    Actions1 ;
    MessagePattern2 [when Guard2] ->
    Actions2 ;
...
    after 0 ->
    ActionsT
end

A timeout of 0 means that the timeout will occur immediately, but Erlang tries all messages currently in the mailbox first.
flush_buffer() ->
  receive
    AnyMessage ->
      flush_buffer()
    after 0 ->
      true
  end.

flush_buffer() completely empties the mailbox of the current actor.
**Priority Messages**

```erlang
priority_receive() ->
    receive
        interrupt -> interrupt
        after 0 ->
        receive
            AnyMessage ->
                AnyMessage
        end
    end.
```

*priority_receive()* will return the first message in the actor’s mailbox, except if there is an *interrupt* message, in which case, *interrupt* will be given priority.

C. Varela
Overview of programming distributed systems

- It is harder than concurrent programming!
- Yet unavoidable in today’s information-oriented society, e.g.:
  - Internet, mobile devices
  - Web services
  - Cloud computing
- Communicating processes with independent address spaces
- Limited network performance
  - Orders of magnitude difference between WAN, LAN, and intra-machine communication.
- Localized heterogeneous resources, e.g., I/O, specialized devices.
- Partial failures, e.g., hardware failures, network disconnection
- Openness: creates security, naming, composability issues.
Universal Actor Names (UAN)

• Consists of *human readable* names.
• Provides location transparency to actors.
• Name to locator mapping updated as actors migrate.
• UAN servers provide mapping between names and locators.
  – Example Universal Actor Name:

    uan://wwc.cs.rpi.edu:3030/cvarela/calendar

    ┌───────────────┐ ┌───────────────┐
    | Name server    | | Unique relative |
    | address and     | | actor name.     |
    | (optional) port.| |                   |
WWC Theaters

C. Varela
Universal Actor Locators (UAL)

- Theaters provide an execution environment for universal actors.
- Provide a layer beneath actors for message passing and migration.
- When an actor migrates, its UAN remains the same, while its UAL changes to refer to the new theater.
- Example Universal Actor Locator:

  \( \text{rmsp://wwc.cs.rpi.edu:4040} \)

  Theater’s IP address and (optional) port.
SALSA Language Support for Worldwide Computing

- SALSA provides linguistic abstractions for:
  - Universal naming (UAN & UAL).
  - Remote actor creation.
  - Location-transparent message sending.
  - Migration.
  - Coordination.

- SALSA-compiled code closely tied to WWC run-time platform.
Universal Actor Creation

- To create an actor locally

  `TravelAgent a = new TravelAgent();`

- To create an actor with a specified UAN and UAL:

  `TravelAgent a = new TravelAgent() at (uan, ual);`

- To create an actor with a specified UAN at current location:

  `TravelAgent a = new TravelAgent() at (uan);`
Message Sending

TravelAgent a = new TravelAgent();

a <- book( flight );

Message sending syntax is the same (<-), independently of actor’s location.
Remote Message Sending

- Obtain a remote actor reference by name.

```java
TravelAgent a = (TravelAgent)
    TravelAgent.getReferenceByName("uan://myhost/ta");

a <- printItinerary();
```
module dcell;

behavior Cell implements ActorService{

    Object content;

    Cell(Object initialContent) {
        content = initialContent;
    }

    Object get() {
        standardOutput <- println ("Returning: "+content);
        return content;
    }

    void set(Object newContent) {
        standardOutput <- println ("Setting: "+newContent);
        content = newContent;
    }

   }

   implements ActorService signals that actors with this behavior are not to be garbage collected.
module dcell;

behavior CellTester {

    void act( String[] args ) {

        if (args.length != 2) {
            standardError <- println(
                "Usage: salsa dcell.CellTester <UAN> <UAL>"");
            return;
        }

        Cell c = new Cell(0) at (new UAN(args[0]), new UAL(args[1]));

        standardOutput <- print( "Initial Value:" ) @
        c <- get() @ standardOutput <- println( token );
    }
}
module dcell;

behavior GetCellValue {

    void act( String[] args ) {
        if (args.length != 1){
            standardOutput <- println(
                "Usage: salsa dcell.GetCellValue <CellUAN>" );
            return;
        }

        Cell c = (Cell) Cell.getReferenceByName(args[0]);

        standardOutput <- print("Cell Value:") @
        c <- get() @
        standardOutput <- println(token);
    }
}

module addressbook;
import java.util.*

behavior AddressBook implements ActorService {

    Hashtable name2email;
    AddressBook() {
        name2email = new HashTable();
    }
    String getName(String email) { ... }
    String getEmail(String name) { ... }
    boolean addUser(String name, String email) { ... }

    void act(String[] args) {
        if (args.length != 0){
            standardOutput<-println("Usage: salsa -Duan=<UAN> -Dual=<UAL> addressbook.AddressBook");
        }
    }
}
module addressbook;

behavior AddUser {
    void act(String[] args) {
        if (args.length != 3) {
            standardOutput<-println("Usage: salsa
                addressbook.AddUser <AddressBookUAN> <Name> <Email>");
            return;
        }
        AddressBook book = (AddressBook)
            AddressBook.getReferenceByName(new UAN(args[0]));
        book<-addUser(args(1), args(2));
    }
}

module addressbook;

behavior GetEmail {
    void act(String[] args) {
        if (args.length != 2) {
            standardOutput <- println("Usage: salsa addressbook.GetEmail <AddressBookUAN> <Name>");
            return;
        }
        getEmail(args(0), args(1));
    }

    void getEmail(String uan, String name) {
        try {
            AddressBook book = (AddressBook) AddressBook.getReferenceByName(new UAN(uan));
            standardOutput <- println(name + "'s email: ") @ book <- getEmail(name) @ standardOutput <- println(token);
        } catch (MalformedUANException e) {
            standardError<-println(e);
        }
    }
}
Erlang Language Support for Distributed Computing

- Erlang provides linguistic abstractions for:
  - Registered processes (actors).
  - Remote process (actor) creation.
  - Remote message sending.
  - Process (actor) groups.
  - Error detection.

- Erlang-compiled code closely tied to Erlang node run-time platform.
• To return our own node name:

    node()  

• To return a list of other known node names:

    nodes()  

• To monitor a node:

    monitor_node(Node, Flag)

If flag is true, monitoring starts. If false, monitoring stops. When a monitored node fails, \{nodedown, Node\} is sent to monitoring process.
Actor Creation

• To create an actor locally

   Agent = spawn(travel, agent, []);

• To create an actor in a specified remote node:

   Agent = spawn(host, travel, agent, []);

   travel is the module name, agent is the function name, Agent is the actor name.

   host is the node name.
Actor Registration

- To register an actor:

\[ \text{register}(ta, \text{Agent}) \]

- To return the actor identified with a registered name:

\[ \text{whereis}(ta) \]

- To remove the association between an atom and an actor:

\[ \text{unregister}(ta) \]

\( ta \) is the registered name (an atom), \( \text{Agent} \) is the actor name (PID).
Message Sending

Agent = spawn(travel, agent, []),
register(ta, Agent)

Agent ! {book, Flight}
ta ! {book, Flight}

Message sending syntax is the same (!) with actor name (Agent) or registered name (ta).
Remote Message Sending

- To send a message to a remote registered actor:

\{ta, host\} ! \{book, Flight\}
Reference Cell Service Example

-module(dcell).
-export([cell/1,start/1]).

cell(Content) ->
  receive
    {set, NewContent} -> cell(NewContent);
    {get, Customer}   -> Customer ! Content,
    cell(Content)
  end.

start(Content) ->
  register(dcell, spawn(dcell, cell, [Content]))
Reference Cell Tester

-module(dcellTester).
-export([main/0]).

main() ->
    dcell:start(0),
    dcell!{get, self()},
    receive
        Value ->
            io:format("Initial Value:~w~n", [Value])
    end.
Reference Cell Client Example

-module(dcellClient).
-export([getCellValue/1]).

catchCellValue(Node) ->
    {dcell, Node}!{get, self()},
    receive
        Value ->
            io:format("Initial Value:\n", [Value])
    end.
Address Book Service

-module(addressbook).
-export([start/0, addressbook/1]).

start() ->
    register(addressbook, spawn(addressbook, addressbook, [[]])).

addressbook(Data) ->
    receive
        {From, {addUser, Name, Email}} ->
            From ! {addressbook, ok},
            addressbook(add(Name, Email, Data));
        {From, {getName, Email}} ->
            From ! {addressbook, getname(Email, Data)},
            addressbook(Data);
        {From, {getEmail, Name}} ->
            From ! {addressbook, getemail(Name, Data)},
            addressbook(Data)
    end.

add(Name, Email, Data) -> ...
getName(Email, Data) -> ...
getemail(Name, Data) -> ...
Address Book Client Example

-module(addressbook_client).
-export([getEmail/1,getName/1,addUser/2]).

addressbook_server() -> 'addressbook@127.0.0.1'.

ggetEmail(Name) -> call_addressbook({getEmail, Name}).
getName(Email) -> call_addressbook({getName, Email}).
addUser(Name, Email) -> call_addressbook({addUser, Name, Email}).

call_addressbook(Msg) ->
    AddressBookServer = addressbook_server(),
    monitor_node(AddressBookServer, true),
    {addressbook, AddressBookServer} ! {self(), Msg},
    receive
        {addressbook, Reply} ->
            monitor_node(AddressBookServer, false),
            Reply;
        {nodedown, AddressBookServer} ->
            no
    end.
Advanced Features of Actor Languages

• SALSA and Erlang support the basic primitives of the actor model:
  – Actors can create new actors.
  – Message passing is asynchronous.
  – State is encapsulated.
  – Run-time ensures fairness.

• SALSA also introduces advanced coordination abstractions: tokens, join blocks, and first-class continuations; SALSA supports distributed systems development including actor mobility and garbage collection. Research projects have also investigated load balancing, malleability (IOS), scalability (COS), and visualization (OverView).

• Erlang introduces a selective receive abstraction to enforce different orders of message delivery, including a timeout mechanism to bypass blocking behavior of receive primitive. Erlang also provides error handling abstractions at the language level, and dynamic (hot) code loading capabilities.
module dcell;

behavior MovingCellTester {

    void act( String[] args ) {

        if (args.length != 3){
            standardError <- println("Usage:
                salsa dcell.MovingCellTester <UAN> <UAL1> <UAL2>");
            return;
        }

        Cell c = new Cell("Hello") at (new UAN(args[0]), new UAL(args[1]));

        standardOutput <- print( "Initial Value:" ) @
        c <- get() @ standardOutput <- println( token ) @
        c <- set("World") @
        standardOutput <- print( "New Value:" ) @
        c <- get() @ standardOutput <- println( token ) @
        c <- migrate(args[2]) @
        c <- set("New World") @
        standardOutput <- print( "New Value at New Location:" ) @
        c <- get() @ standardOutput <- println( token );
    }
}

C. Varela
Address Book Migrate Example

```java
module addressbook;

behavior MigrateBook {
    void act(String[] args) {
        if (args.length != 2) {
            standardOutput<-println("Usage: salsa addressbook.MigrateBook <AddressBookUAN> <NewUAL>");
            return;
        }
        AddressBook book = (AddressBook)
            AddressBook.getReferenceByName(new UAN(args[0]));
        book<-migrate(args(1));
    }
}
```
Actor Garbage Collection

- Implemented since SALSA 1.0 using pseudo-root approach.
- Includes distributed cyclic garbage collection.
- For more details, please see:


Actor GC vs. Object GC

Actor Reference Graph

Passive Object Reference Graph

Carlos Varela
IOS: Load Balancing and Malleability

- **Middleware**
  - A software layer between distributed applications and operating systems.
  - Alleviates application programmers from directly dealing with distribution issues
    - Heterogeneous hardware/O.S.s
    - Load balancing
    - Fault-tolerance
    - Security
    - Quality of service

- **Internet Operating System (IOS)**
  - A decentralized framework for adaptive, scalable execution
  - Modular architecture to evaluate different distribution and reconfiguration strategies


Component Malleability

- New type of reconfiguration:
  - Applications can dynamically change component granularity
- Malleability can provide many benefits for HPC applications:
  - Can more adequately reconfigure applications in response to a dynamically changing environment:
    - Can scale application in response to dynamically joining resources to improve performance.
    - Can provide soft fault-tolerance in response to dynamically leaving resources.
  - Can be used to find the ideal granularity for different architectures.
  - Easier programming of concurrent applications, as parallelism can be provided transparently.
Component Malleability

- Modifying application component granularity dynamically (at runtime) to improve scalability and performance.
- SALSA-based malleable actor implementation.
- MPI-based malleable process implementation.
- IOS decision module to trigger split and merge reconfiguration.
- For more details, please see:

Distributed Systems Visualization

- Generic online Java-based distributed systems visualization tool
- Uses a declarative Entity Specification Language (ESL)
- Instruments byte-code to send events to visualization layer.
- For more details, please see:

Open Source Code

- Consider to contribute!
- Visit our web pages for more info:
  - SALSA: http://wcl.cs.rpi.edu/salsa/
  - IOS: http://wcl.cs.rpi.edu/ios/
  - OverView: http://wcl.cs.rpi.edu/overview/
  - COS: http://wcl.cs.rpi.edu/cos/
  - PILOTS: http://wcl.cs.rpi.edu/pilots/
  - MilkyWay@Home: http://milkyway.cs.rpi.edu/
Erlang Language Support for Fault-Tolerant Computing

- Erlang provides linguistic abstractions for:
  - Error detection.
    - Catch/throw exception handling.
    - Normal/abnormal process termination.
    - Node monitoring and exit signals.
  - Process (actor) groups.
  - Dynamic (hot) code loading.
Exception Handling

• To protect sequential code from errors:

```java
    catch Expression
```

If failure does not occur in expression evaluation, catch Expression returns the value of the expression.

• To enable non-local return from a function:

```java
    throw({ab_exception, user_exists})
```
Address Book Example

-module(addressbook).
-export([start/0,addressbook/1]).

start() ->
    register(addressbook, spawn(addressbook, addressbook, [[]])).

addressbook(Data) ->
    receive
        {From, {addUser, Name, Email}} ->
            From ! {addressbook, ok},
            addressbook(add(Name, Email, Data));
    end.

add(Name, Email, Data) ->
    case getemail(Name, Data) of
        undefined ->
            [{Name,Email}|Data];
        _ -> % if Name already exists, add is ignored.
            Data
    end.

getemail(Name, Data) -> ...

C. Varela
Address Book Example with Exception

addressbook(Data) ->
    receive
        {From, {addUser, Name, Email}} ->
            case catch add(Name, Email, Data) of
                {ab_exception, user_exists} ->
                    From ! {addressbook, no},
                    addressbook(Data);
                NewData->
                    From ! {addressbook, ok},
                    addressbook(NewData)
            end;
    end;

    ...
end.

add(Name, Email, Data) ->
    case getemail(Name, Data) of
        undefined ->
            undefined ->
                [{Name,Email}|Data];
            _ ->       % if Name already exists, exception is thrown.
                throw({ab_exception, user_exists})
        end.
Normal/abnormal termination

• To terminate an actor, you may simply return from the function the actor executes (without using tail-form recursion). This is equivalent to calling:
  \[
  \text{exit(normal)}.
  \]

• Abnormal termination of a function, can be programmed:
  \[
  \text{exit({ab_error, no_msg_handler})}
  \]
  equivalent to:
  \[
  \text{throw({'EXIT',{ab_error, no_msg_handler}})}
  \]

• Or it can happen as a run-time error, where the Erlang run-time sends a signal equivalent to:
  \[
  \text{exit(badarg) \quad \% Wrong argument type}
  \text{exit(function_clause) \quad \% No pattern match}
  \]
Address Book Example with Exception and Error Handling

addressbook(Data) ->
    receive
        {From, {addUser, Name, Email}} ->
            case catch add(Name, Email, Data) of
                {ab_exception, user_exists} ->
                    From ! {addressbook, no},
                    addressbook(Data);
                {ab_error, What} -> … % programmer-generated error (exit)
                {'EXIT', What} -> … % run-time-generated error
                NewData->
                    From ! {addressbook, ok},
                    addressbook(NewData)
            end;
        …
    end.

C. Varela
Node monitoring

- To monitor a node:

```python
monitor_node(Node, Flag)
```

If `flag` is true, monitoring starts. If false, monitoring stops. When a monitored node fails, `{nodedown, Node}` is sent to monitoring process.
Address Book Client Example with Node Monitoring

-module(addressbook_client).
-export([getEmail/1,getName/1,addUser/2]).

addressbook_server() -> 'addressbook@127.0.0.1'.

getEmail(Name) -> call_addressbook({getEmail, Name}).
getName(Email) -> call_addressbook({getName, Email}).
addUser(Name, Email) -> call_addressbook({addUser, Name, Email}).

call_addressbook(Msg) ->
    AddressBookServer = addressbook_server(),
    monitor_node(AddressBookServer, true),
    {addressbook, AddressBookServer} ! {self(), Msg},
    receive
        {addressbook, Reply} ->
            monitor_node(AddressBookServer, false),
            Reply;
        {nodedown, AddressBookServer} ->
            no
    end.
Process (Actor) Groups

• To create an actor in a specified remote node:

\[
\text{Agent} = \text{spawn}(\text{host, travel, agent, []});
\]

• To create an actor in a specified remote node and create a link to the actor:

\[
\text{Agent} = \text{spawn\_link}(\text{host, travel, agent, []});
\]

An 'EXIT' signal will be sent to the originating actor if the host node does not exist.
Group Failure

• Default error handling for linked processes is as follows:
  – Normal exit signal is ignored.
  – Abnormal exit (either programmatic or system-generated):
    • Bypass all messages to the receiving process.
    • Kill the receiving process.
    • Propagate same error signal to links of killed process.

• All linked processes will get killed if a participating process exits abnormally.
Dynamic code loading

- To update (module) code while running it:

```prolog
-module(m).
-export([loop/0]).

loop() ->
  receive
    code_switch ->
      m:loop();
    Msg -> ...
      loop()
  end.
```

code_switch message dynamically loads the new module code. Notice the difference between `m:loop()` and `loop()`.
Declarative operations (1)

• An operation is *declarative* if whenever it is called with the same arguments, it returns the same results independent of any other computation state.

• A declarative operation is:
  – *Independent* (depends only on its arguments, nothing else)
  – *Stateless* (no internal state is remembered between calls)
  – *Deterministic* (call with same operations always give same results)

• Declarative operations can be composed together to yield other declarative components
  – All basic operations of the declarative model are declarative and combining them always gives declarative components.
Declarative operations (2)
Why declarative components (1)

• There are two reasons why they are important:
• *(Programming in the large)* A declarative component can be written, tested, and proved correct independent of other components and of its own past history.
  – The complexity (reasoning complexity) of a program composed of declarative components is the *sum* of the complexity of the components
  – In general the reasoning complexity of programs that are composed of nondeclarative components explodes because of the intimate interaction between components
• *(Programming in the small)* Programs written in the declarative model are much easier to reason about than programs written in more expressive models (e.g., an object-oriented model).
  – Simple algebraic and logical reasoning techniques can be used
Why declarative components (2)

• Since declarative components are mathematical functions, algebraic reasoning is possible i.e. substituting equals for equals

• The declarative model of CTM Chapter 2 guarantees that all programs written are declarative

• Declarative components can be written in models that allow stateful data types, but there is no guarantee

Given

\[ f(a) = a^2 \]

We can replace \( f(a) \) in any other equation

\[ b = 7f(a)^2 \text{ becomes } b = 7a^4 \]
The word *declarative* means many things to many people. Let’s try to eliminate the confusion.

The basic intuition is to program by defining the *what* without explaining the *how*.
Oz kernel language

The following defines the syntax of a statement, \( \langle s \rangle \) denotes a statement

\[
\langle s \rangle ::= \text{skip}
\]

\[
| \langle x \rangle = \langle y \rangle
\]

\[
| \langle x \rangle = \langle v \rangle
\]

\[
| \langle s_1 \rangle \langle s_2 \rangle
\]

\[
| \text{local} \, \langle x \rangle \, \text{in} \, \langle s_1 \rangle \, \text{end}
\]

\[
| \text{proc} \, \{ \langle x \rangle \, \langle y_1 \rangle \ldots \langle y_n \rangle \} \, \langle s_1 \rangle \, \text{end}
\]

\[
| \text{if} \, \langle x \rangle \, \text{then} \, \langle s_1 \rangle \, \text{else} \, \langle s_2 \rangle \, \text{end}
\]

\[
| \{ \langle x \rangle \, \langle y_1 \rangle \ldots \langle y_n \rangle \}
\]

\[
| \text{case} \, \langle x \rangle \, \text{of} \, \langle \text{pattern} \rangle \, \text{then} \, \langle s_1 \rangle \, \text{else} \, \langle s_2 \rangle \, \text{end}
\]

empty statement

variable-variable binding

variable-value binding

sequential composition

declaration

procedure introduction

conditional

procedure application

pattern matching
Why the Oz KL is declarative

- All basic operations are declarative
- Given the components (sub-statements) are declarative,
  - sequential composition
  - local statement
  - procedure definition
  - procedure call
  - if statement
  - case statement

are all declarative (independent, stateless, deterministic).
What is state?

- State is a sequence of values in time that contains the intermediate results of a desired computation.
- Declarative programs can also have state according to this definition.
- Consider the following program:

```plaintext
fun {Sum Xs A}
  case Xs
  of X|Xr then {Sum Xr A+X}
  [] nil then A
  end
end

{Browse {Sum [1 2 3 4] 0}}
```
What is implicit state?

The two arguments Xs and A represent an implicit state

<table>
<thead>
<tr>
<th>Xs</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1 2 3 4]</td>
<td>0</td>
</tr>
<tr>
<td>[2 3 4]</td>
<td>1</td>
</tr>
<tr>
<td>[3 4]</td>
<td>3</td>
</tr>
<tr>
<td>[4]</td>
<td>6</td>
</tr>
<tr>
<td>nil</td>
<td>10</td>
</tr>
</tbody>
</table>

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

{Browse \{Sum [1 2 3 4] 0\}}
What is explicit state: Example?

An unbound variable

A cell $C$ is created with initial value 5
$X$ is bound to $C$

The cell $C$, which $X$ is bound to, is assigned the value 6
What is explicit state: Example?

An unbound variable

A cell \( C \) is created with initial value 5

\( X \) is bound to \( C \)

The cell \( C \), which \( X \) is bound to, is assigned the value 6

- The cell is a value container with a unique **identity**
- \( X \) is really bound to the **identity** of the cell
- When the cell is assigned, \( X \) does not change
What is explicit state?

- **X = {NewCell I}**
  - Creates a cell with initial value I
  - Binds X to the identity of the cell
- **Example: X = {NewCell 0}**
- **{Assign X J}**
  - Assumes X is bound to a cell C (otherwise exception)
  - Changes the content of C to become J
- **Y = {Access X}**
  - Assumes X is bound to a cell C (otherwise exception)
  - Binds Y to the value contained in C
The stateful model

\[ \langle s \rangle ::= \text{skip} \]
\[ \langle s_1 \rangle \langle s_2 \rangle \]
\[ \ldots \]
\[ \{ \text{NewCell} \langle x \rangle \langle c \rangle \} \]
\[ \{ \text{Exchange} \langle c \rangle \langle x \rangle \langle y \rangle \} \]

Empty statement
Statement sequence
Cell creation
Cell exchange

Exchange: bind \( \langle x \rangle \) to the old content of \( \langle c \rangle \) and set the content of the cell \( \langle c \rangle \) to \( \langle y \rangle \)
The stateful model

<table>
<thead>
<tr>
<th></th>
<th>{NewCell ⟨x⟩ ⟨c⟩}</th>
<th>cell creation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{Exchange ⟨c⟩ ⟨x⟩ ⟨y⟩}</td>
<td>cell exchange</td>
</tr>
</tbody>
</table>

Exchange: bind ⟨x⟩ to the old content of ⟨c⟩ and set the content of the cell ⟨c⟩ to ⟨y⟩

    proc {Assign C X} {Exchange C _ X} end

    fun {Access C} X in {Exchange C X X} X end

    C := X is syntactic sugar for {Assign C X}
    @C is syntactic sugar for {Access C}
    X=C:=Y is syntactic sugar for {Exchange C X Y}
Abstract data types (revisited)

- For a given functionality, there are many ways to package the ADT. We distinguish three axes.
- **Open vs. secure ADT**: is the internal representation visible to the program or hidden?
- **Declarative vs. stateful ADT**: does the ADT have encapsulated state or not?
- **Bundled vs. unbundled ADT**: is the data kept together with the operations or is it separable?
- Let us see what our stack ADT looks like with some of these possibilities
Stack:
Secure, stateful, and *bundled*

- This is the simplest way to make a secure stateful stack:
  
  ```
      C={NewCell nil}
  in
      proc {Push X} {Assign C X|{Access C}} end
      fun {Pop} case {Access C} of X|S then {Assign C S} X end end
      fun {IsEmpty} {Access C} ==nil end
  end
  ```

- Compare the declarative with the stateful versions: the declarative version needs two arguments per operation, the stateful version uses higher-order programming (instantiation)

- With some syntactic support, this is *object-based programming*
Four ways to package a stack

- **Open, declarative, and unbundled**: the usual declarative style, e.g., in Prolog and Scheme
- **Secure, declarative, and unbundled**: use wrappers to make the declarative style secure
- **Secure, stateful, and unbundled**: an interesting variation on the usual object-oriented style
- **Secure, stateful, and bundled**: the usual object-oriented style, e.g., in Smalltalk and Java
- **Other possibilities**: there are four more possibilities!

Exercise: Try to write all of them.
Object-oriented programming

• Supports
  – Encapsulation
  – Compositionality
  – Instantiation

• Plus
  – Inheritance
Inheritance

• Programs can be built in hierarchical structure from ADT’s that depend on other ADT’s (Components)
• Object-oriented programming (inheritance) is based on the idea that ADTs have so much in common
• For example, sequences (stacks, lists, queues)
• Object oriented programming enables building ADTs incrementally, through inheritance
• An ADT can be defined to inherit from another abstract data type, substantially sharing functionality with that abstract data type
• Only the difference between an abstract datatype and its ancestor has to be specified
What is object-oriented programming?

• OOP (Object-oriented programming) = encapsulated state + inheritance

• Object
  – An entity with unique identity that encapsulates state
  – State can be accessed in a controlled way from outside
  – The access is provided by means of methods (procedures that can directly access the internal state)

• Class
  – A specification of objects in an incremental way
  – Incrementality is achieved inheriting from other classes by specifying how its objects (instances) differ from the objects of the inherited classes
## Instances (objects)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interface</strong> (what methods are available)</td>
<td></td>
</tr>
<tr>
<td><strong>State</strong> (attributes)</td>
<td></td>
</tr>
<tr>
<td><strong>Procedures</strong> (methods)</td>
<td></td>
</tr>
</tbody>
</table>
Classes (simplified syntax)

A class is a statement

class 〈ClassVariable〉
    attr
        〈AttrName1〉
        :
        〈AttrNameN〉
    meth 〈Pattern1〉 〈Statement〉 end
        :
    meth 〈PatternN〉 〈Statement〉 end
end
Classes in Oz

The class Counter has the syntactic form

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

• An object is created from a class using the procedure `New/3`, whose first argument is the class, the second is the initial method, and the result is the object (such as in the functor and procedure approaches).

• `New/3` is a generic procedure for creating objects from classes.

```declare C = {New Counter init(0)}
{C display}
{C inc(1)}
{C display}```
• A class $X$ is defined by:
  - `class X ... end`

• Attributes are defined using the attribute-declaration part before the method-declaration part:
  - `attr A_1 ... A_N`

• Then follows the method declarations, each has the form:
  - `meth E S end`

• The expression $E$ evaluates to a method head, which is a record whose label is the method name.
Summary

• An attribute \( A \) is accessed using \( @A \).
• An attribute is assigned a value using \( A := E \).
• A class can be defined as a value:
  • \( X = \texttt{class} \$ \ldots \texttt{end} \).
Classes as incremental ADTs

• Object-oriented programming allows us to define a class by extending existing classes

• **Three things have to be introduced**
  – How to express inheritance, and what does it mean?
  – How to access particular methods in the new class and in preexisting classes
  – Visibility – what part of the program can see the attributes and methods of a class

• The notion of delegation as a substitute for inheritance
Inheritance

- Inheritance should be used as a way to specialize a class while retaining the relationship between methods.
- In this way it is just an extension of an ADT.
- The other view is that inheritance is just a (lazy) way to construct new abstract data types!
- No relationships are preserved.
Inheritance

```ruby
class Account
  attr balance: 0
  meth transfer(Amount)
    balance := @balance + Amount
  end
  meth getBal(B)
    B = @balance
  end
end

A = {New Account transfer(100)}
```

C. Varela; Adapted from S. Haridi and P. Van Roy
Conservative extension

class VerboseAccount

    from Account

    meth verboseTransfer(Amount)

    ...

end

end

The class VerboseAccount has the methods: transfer, getBal, and verboseTransfer
Non-Conservative extension

```ruby
class AccountWithFee
  from VerboseAccount
  attr fee: 5
  meth transfer(amount)
    ...
  end
end
```

The class AccountWithFee has the methods: transfer, getBal, and verboseTransfer. The method transfer has been redefined (overridden) with another definition.
Inheritance II

Non-Conservative extension

class AccountWithFee
  from VerboseAccount
  attr fee:5
  meth transfer(Amount)
  ...
end
end
Polymorphism

The ability for operations to take objects (instances) of different types.

For example, the transfer method can be invoked in account object instances of three different classes.

The most specific behavior should be executed.
Static and dynamic binding

Dynamic binding

• Inside an object O we want to invoke a method M

• This is written as \{\texttt{self} \, M\}, and chooses the method visible in the current object (M of D)
Static and dynamic binding

Static binding

• Inside an object O we want to invoke a method M in a specific (super) class
• This is written as C, M and chooses the method visible in the super class C (M of C)
Static method calls

- Given a class and a method head $m(...)$, a static method-call has the following form:
  \[C, m(...)]\]

- Invokes the method defined in the class argument.

- A static method call can only be used inside class definitions.

- The method call takes the current object denoted by `self` as implicit argument.

- The method $m$ could be defined in the class $C$, or inherited from a super class.
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.
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}
```
State

- How to make a function learn from its past?
- We would like to add memory to a function to remember past results
- Adding memory as well as concurrency is an essential aspect of modeling the real world
- Consider \{FastPascal N\}: we would like it to remember the previous rows it calculated in order to avoid recalculating them
- We need a concept (memory cell) to store, change and retrieve a value
- The simplest concept is a (memory) cell which is a container of a value
- One can create a cell, assign a value to a cell, and access the current value of the cell
- Cells are not variables

```declare
C = \{NewCell 0\}
\{Assign C \{Access C\} + 1\}
\{Browse \{Access C\}\}
```
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

C = {NewCell 0}
cell C contains 0

{Assign C 1}
cell C contains 1

{Assign C 2}
cell C contains 2 (final value)
```

S. Haridi and P. Van Roy
Nondeterminism (3)

```
declare
C = {NewCell 0}

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

```
t₀  C = {NewCell 0}
cell C contains 0

{Assign C 2}
cell C contains 2

{Assign C 1}
cell C contains 1 (final value)
```
Nondeterminism (4)

```plaintext
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

```plaintext
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
```

```
t_0
C = {NewCell 0}

I = {Access C}
I equal 0

J = {Access C}
J equal 0

{Assign C J+1}
C contains 1

{Assign C I+1}
C contains 1

t_4

C contains 1
```

S. Haridi and P. Van Roy
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 thousands of times too high, resulting in death or injury)

1. If possible try to avoid concurrency and state together
2. Encapsulate state and communicate between threads using dataflow
3. 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
  sequence of ops 1
end

Thread 1

lock L then
  sequence of ops 2
end

Thread 2
```
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
Locks and Deadlock: Dining Philosophers
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.
Declarative Concurrency

• Declarative concurrency is for programs with no observable nondeterminism, the result is a function
• Independent procedures that execute on their pace and may communicate through shared dataflow variables
Single-assignment Variables

• Variables are short-cuts for values, they cannot be assigned more than once
  
  ```
  declare
  V = 9999*9999
  {Browse V*V}
  ```

• Variable identifiers: is what you type
• Store variable: is part of the memory system
• The `declare` statement creates a store variable and assigns its memory address to the identifier ’V’ in the environment
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

```plaintext
declare X
thread
  {Delay 5000} X=99
end
{Browse ‘Start’} {Browse X*X}
```

```plaintext
declare X
thread
  {Browse ‘Start’} {Browse X*X}
end
{Delay 5000} X=99
```
The concurrent model

Multiple semantic stacks (threads)

Semantic Stack 1

Semantic Stack N

\[ w = a \]
\[ z = \text{person(age: } y) \]
\[ x \]
\[ y = 42 \]
\[ u \]

Single-assignment store
Concurrent declarative model

The following defines the syntax of a statement, $\langle s \rangle$ denotes a statement

$$\langle s \rangle ::= \text{skip} \quad \text{empty statement}$$
$$\quad \langle x \rangle = \langle y \rangle \quad \text{variable-variable binding}$$
$$\quad \langle x \rangle = \langle v \rangle \quad \text{variable-value binding}$$
$$\quad \langle s_1 \rangle \langle s_2 \rangle \quad \text{sequential composition}$$
$$\quad \text{local } \langle x \rangle \text{ in } \langle s_1 \rangle \text{ end} \quad \text{declaration}$$
$$\quad \text{proc } \{ \langle x \rangle \langle y_1 \rangle \ldots \langle y_n \rangle \} \langle s_1 \rangle \text{ end} \quad \text{procedure introduction}$$
$$\quad \text{if } \langle x \rangle \text{ then } \langle s_1 \rangle \text{ else } \langle s_2 \rangle \text{ end} \quad \text{conditional}$$
$$\quad \{ \langle x \rangle \langle y_1 \rangle \ldots \langle y_n \rangle \} \quad \text{procedure application}$$
$$\quad \text{case } \langle x \rangle \text{ of } \langle \text{pattern} \rangle \text{ then } \langle s_1 \rangle \text{ else } \langle s_2 \rangle \text{ end} \quad \text{pattern matching}$$
$$\quad \text{thread } \langle s_1 \rangle \text{ end} \quad \text{thread creation}$$
The concurrent model

Top of Stack, Thread i

Single-assignment store
The concurrent model

Top of Stack, Thread i \rightarrow ST \langle s_i \rangle_E

Single-assignment store
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
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 or multiple compatible bindings (e.g., to partial values), 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
Example of runnable threads

\[\text{proc}\ \{\text{Loop } P\ N\}\]
\[
\text{if } N > 0 \text{ then}
\]
\[
\{P\} \{\text{Loop } P\ N-1\}
\]
\[
\text{else skip end}
\]
end

thread \{\text{Loop}
\[
\text{proc}\ \{\$\} \{\text{Show 1}\} \text{ end}
\]
1000\}
end

thread \{\text{Loop}
\[
\text{proc}\ \{\$\} \{\text{Show 2}\} \text{ end}
\]
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.

```
decare X
{Browse X}
local Y in
  thread {Delay 1000} Y = 10*10 end
  X = Y + 100*100
end
```
Illustrating dataflow computation

```latex
\begin{align*}
declare & \ X0 \ X1 \ X2 \ X3 \\
\& \ {\{\text{Browse} \ [X0 \ X1 \ X2 \ X3]\}} \\
\text{thread} & \quad Y0 \ Y1 \ Y2 \ Y3 \\
in & \quad \{\text{Browse} \ [Y0 \ Y1 \ Y2 \ Y3]\} \\
\quad Y0 &= X0 + 1 \\
\quad Y1 &= X1 + Y0 \\
\quad Y2 &= X2 + Y1 \\
\quad Y3 &= X3 + Y2 \\
\& \ {\{\text{Browse} \ \text{completed}\}} \\
\end{align*}
```

- 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 \{\text{Map } Xs \ F\} \\
\hspace{1em} \text{case } Xs \\
\hspace{2em} \text{of } \text{nul} \hspace{0.5em} \text{then } \text{nul} \\
\hspace{3em} [] X|Xr \hspace{0.5em} \text{then thread } \{\text{F } X\} \hspace{0.5em} \text{end} \hspace{0.5em} |\{\text{Map } Xr \ F\} \\
\hspace{2em} \text{end} \\
\hspace{1em} \text{end}

• What this looks like in the kernel language:

proc \{\text{Map } Xs \ F \ Rs\} \\
\hspace{1em} \text{case } Xs \\
\hspace{2em} \text{of } \text{nul} \hspace{0.5em} \text{then } Rs = \text{nul} \\
\hspace{3em} [] X|Xr \hspace{0.5em} \text{then } R \ Rr \hspace{0.5em} \text{in} \\
\hspace{4em} Rs = R|Rr \\
\hspace{5em} \text{thread } \{\text{F } X \ R\} \hspace{0.5em} \text{end} \\
\hspace{4em} |\{\text{Map } Xr \ F \ Rr\} \\
\hspace{2em} \text{end} \\
\hspace{1em} \text{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
fun \{Fib \ X\} 
    if \ X\leq2 \ then \ 1 
    else \ F1 \ F2 \ in 
        \begin{align*} 
        F1 & = \text{thread} \ \{\text{Fib} \ X-1\} \ \text{end} \\
        F2 & = \{\text{Fib} \ X-2\} 
        \end{align*}
    \end{align*}
\end{align*}
\begin{align*} 
F1 + F2 
\end{align*}
\text{end} 
\text{end} 
\begin{align*} 
\text{Dataflow dependency} 
\end{align*}
Execution of \{\text{Fib 6}\}
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 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 II

- **Producer**, produces incrementally the elements
- **Transducer(s)**, transform(s) the elements of the stream
- **Consumer**, accumulates the results
Stream communication patterns

• The producer, transducers, and the consumer can, in general, be described by certain program patterns
• We show various patterns
fun {Producer State}
    if {More State} then
        X = {Produce State} in
        X | {Producer {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 Producer

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

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

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

fun \{\text{Consumer State InStream}\}
\begin{align*}
\text{case } \text{InStream} & \text{ of } \text{nil} \text{ then } \{\text{Final State}\} \\
& \text{ \quad } \text{\quad } \begin{cases} \\
\text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{The consumer suspends until} \\
\text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{InStream is either a cons or a nil} \\
\end{cases} \\
\text{\quad } \text{\quad } \{\text{Consume X State}\} \text{ in} \\
\text{\quad } \text{\quad } \{\text{Consumer NextState RestInStream}\} \\
\end{align*} \\
\text{end} \\
\text{end} \\
\text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{\quad } \text{Final and Consume are problem dependent}

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

Consume takes X and State \(\Rightarrow X + \text{State}\)
Transducer Pattern 1

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

• A transducer keeps its state in State, receives messages on InStream and sends messages on OutStream
fun {Transducer State InStream}
    case InStream
    of nil then nil
    [] X | RestInStream then
        if {Test X#State} then
            NextState#TX = {Transform X State}
            TX | {Transducer NextState RestInStream}
        else {Transducer State RestInStream} end
    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
```

Filter is a transducer that takes an Instream and incrementally produces an Outstream that satisfies the predicate F.

```
local Xs Ys in
thread Xs = {Generate 1 100} end
thread Ys = {Filter Xs IsOdd} end
thread {Browse Ys} end
end
```
Larger example: The sieve of Eratosthenes

- Produces prime numbers
- It takes a stream 2...N, peals off 2 from the rest of the stream
- Delivers the rest to the next sieve
fun {Sieve Xs}
    case Xs
    of nil then nil
    nil X|Xr then Ys in
        thread Ys = {Filter Xr fun {\$ Y} Y mod X \neq 0 end} end
        X | {Sieve Ys}
    end
end

• The program forks a filter thread on each sieve call
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
end

7 | 11 |...
Concurrent control abstraction

• We have seen how threads are forked by ’thread ... end’
• A natural question to ask is: how can we 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⟩ X_n = X_{n-1} end
  {Wait Xn}
  % Continue main thread
  ```

- When all threads terminate the variables $X_1 \ldots X_N$ will be merged together labeling a single box that contains the value `unit`.

- `{Wait X_N}` suspends the main thread until $X_N$ is bound.
Concurrent Composition

\[
\text{conc } S_1 [] S_2 [] \ldots [] S_n \text{ end}
\]

\{Conc \begin{align*}
\text{[ proc}\{\}$ & S1 \text{ end} \\
\text{proc}\{\}$ & S2 \text{ end} \\
\ldots \\
\text{proc}\{\}$ & S_n \text{ end} \end{align*} \}

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

```plaintext
local
  proc {Ping N}
    for I in 1..N do
      {Delay 500} {Browse ping}
    end
    {Browse 'ping terminate'}
  end
proc {Pong N}
  for I in 1..N do
    {Delay 600} {Browse pong}
  end
  {Browse 'pong terminate'}
end
in .... end

local
  ....
in
  {Browse 'game started'}
  {Conc
    [ proc {$_} {Ping 1000} end
      proc {$_} {Pong 1000} end ]
  }
  {Browse 'game terminated'}
end
```
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 `!!` 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

- **declare** Y
  {ByNeed fun {$} 1 end Y}
  {Browse Y}

- we will observe that Y becomes a future, i.e. we will see Y<Future> in the Browser.

- 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 {Wait Y} which triggers the producing procedure.
Summary 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.