LINFO1104 – LSINC1104
Concepts, paradigms, and semantics of programming languages

Lectures 10 & 11
Limitations of deterministic dataflow and message-passing concurrency

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview of lectures 10-11

- Limitation of deterministic dataflow
  - Some applications cannot be written in deterministic dataflow! We explain why not.
- Two new paradigms beyond deterministic dataflow
  - We overcome the limitation by adding one new concept, ports, to deterministic dataflow.
  - We define a new paradigm, message-passing concurrency. This paradigm uses port objects and active objects, which are both agents defined with ports.
  - We define a second paradigm, deterministic dataflow with ports. This paradigm does not use agents but uses deterministic dataflow as much as possible (like we did in the previous lecture) and adds ports only when they are necessary.

2

Limitations of deterministic dataflow

3

Limitations of deterministic dataflow

- In lectures 8-9 we saw deterministic dataflow, which makes concurrent programming very easy
  - It allows “Concurrency for Dummies”: threads can be added to the program at will without changing the result
- But unfortunately it cannot be used all the time!
  - It has a strong limitation: it cannot be used to write programs when the nondeterminism must be visible
  - But why must nondeterminism sometimes be visible? Let’s see an example: a client/server application.

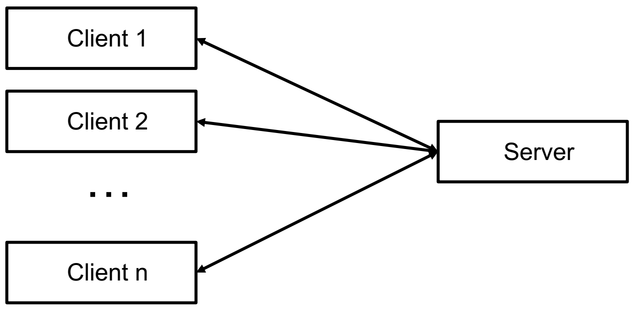
4

Client/server application (1)

- A client/server application consists of a set of clients all communicating to one server
  - The clients and the server are concurrent agents
  - Each client sends messages to the server and receives replies
- Client/server applications are ubiquitous on the Internet
  - For example, all Web stores are client/servers: the users are clients and the store is the server
  - When shopping at Amazon, your Web browser sends messages and receives replies from the Amazon server
- Client/server cannot be written in deterministic dataflow!
  - Why not? Let's try and see what goes wrong! Try it yourself!

5

Client/server application (2)



- Each client has a link to the server and can send messages to the server at any time
- The server receives each message, does a local computation, and then replies immediately

6

Client/server: first attempt

- Let's try to write a client/server in deterministic dataflow
  - Assume that there are two clients, each with an output stream, and the server receives both
- Here is the server code:

This doesn't work!
Why not?

```
proc {Server S1 S2}
    case S1|S2 of (M1|T1)|S2 then
            (handle M1) {Server T1 S2}
    [] S1|(M2|T2) then
            (handle M2) {Server S1 T2}
    end
end
```

7

Client/server: second attempt

- The first attempt does not work if Client 2 sends a message and Client 1 sends nothing
- We can try doing it the other way around:

```
proc {Server S1 S2}
    case S1|S2 of S1|(M2|T2) then
            (handle M1) {Server S1 T2}
    [] (M1|T1)|S2 then
            (handle M2) {Server T1 S2}
    end
end
```

- This doesn't work if Client 1 sends a message and Client 2 sends nothing!

8

Client/server: third attempt

- Maybe the server has to receive from both clients:

```
proc {Server S1 S2}
     case S1|S2 of (M1|T1)|(M2|T2) then
             (handle M1)
             (handle M2)
             {Server T1 T2}
     end
end
```

- This does not work either! (Why not?)

9

What is the problem?

- The **case** statement waits on a single pattern
  - This is because of determinism: with the same input, the **case** statement must give the same result
- But the server must wait on two patterns
  - Either M1 from Client 1 or M2 from Client 2
  - Either pattern is possible, it depends on when each client sends the message and on how long the message takes to reach the server
    - The decision is made outside the program
  - This means exactly that execution is nondeterministic!

10

Understanding nondeterminism

- Nondeterminism means that a choice is made outside of the program's control
  - This is exactly what is happening here: the choice is the arrival order of the client messages, which depends on the human clients and on the message travel time
- The nondeterminism is inherently part of the client/server execution, it cannot be avoided
  - The nondeterminism is a consequence of the initial requirement: "The server receives each message, does a local computation, and then replies immediately"
  - This means that the reply cannot be delayed while the server waits for another message

11

Overcoming the limitation

12

Overcoming the limitation

- Deterministic dataflow cannot express an application that requires nondeterminism
- To do this, we need to extend the kernel language with a new concept
- The new concept must be able to wait on several events nondeterministically
  - The new language is no longer deterministic!
- We will show two possible solutions

13

Solution 1: WaitTwo

- We introduce the function:
  {WaitTwo X Y}
  with the following semantics:
  - {WaitTwo X Y} can return 1 if X is bound
  - {WaitTwo X Y} can return 2 if Y is bound
  - If either X or Y is bound, {WaitTwo X Y} will return
- If both X and Y are unbound, it just waits
- If both X and Y are bound, it can return either 1 or 2, both are possible (nondeterminism!)

14

Client/Server with WaitTwo

- Here is the client/server with WaitTwo:

```
proc {Server S1 S2}
     C={WaitTwo S1 S2}
in
     case C|S1|S2 of 1|(M1|T1)|S2 then
            (handle M1) {Server T1 S2}
     [] 2|S1|(M2|T2) then
            (handle M2) {Server S1 T2}
     end
end
```

- If Client 1 sends a message, C=1 and it is handled
- If Client 2 sends a message, C=2 and it is handled
- What happens if both Client 1 and Client 2 send messages?

15

WaitTwo is not scalable

- What happens if we have millions of clients?
  - WaitTwo solves the problem for two clients
  - How can we wait on millions of clients?
- One possibility is to “merge” all client streams into a single stream:

```
fun {Merge S1 S2}
     C={WaitTwo S1 S2}
in
     case C|S1|S2 of 1|(M1|T1)|S2 then M1|{Merge T1 S2}
     [] 2|S1|(M2|T2) then M2|{Merge S1 T2}
     end
end
```

- With Merge we build a huge tree of stream mergers. It must expand and contract if new clients arrive or old clients leave. Not very nice!

16

Solution 2: Ports

- A better solution is to add ports (named streams)
- Ports have two operations:

```
P={NewPort S} % Create port P with stream S
{Send P X}     % Add X to end of port P's stream
```

- How does this solve our problem?
  - With a million clients C_1 to $C_{1000000}$:
    Each client C_i does {Send M_i P} for each message it sends
  - The server reads the stream S, which contains all messages from all clients in some nondeterministic order

17

Port example operations

- We create a port and do sends:

```
P={NewPort S}
{Browse S} % Displays _
{Send P a}  % Displays a|_
{Send P b}  % Displays a|b|_
```

- What happens if we do:

```
thread {Send P c} end
thread {Send P d} end
```

- What are the possible results of these two sends for all choices of the scheduler?

18

Port semantics (1)

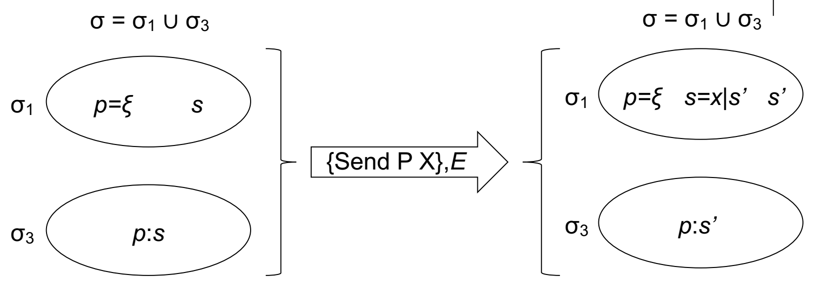
- Assume single-assignment store σ_1 with variables
- Assume a port store σ_3 that contains pairs of variables
  - (Remember σ_2 was the cell store we introduced before)

environment

- P={NewPort S}, {P→*p*, S→*s*}
  - Assume unbound variables *p*, *s* ∈ σ_1
  - Create fresh name ξ, bind *p*=ξ, add pair *p*:*s* to σ_2
- {Send P X}, {P→*p*, X→*x*}
  - Assume *p*=ξ, unbound variable *s* ∈ σ_1 , *p*:*s* ∈ σ_2
  - Create fresh unbound variable *s'*, bind *s*=*x*|*s'*, update pair to *p*:*s'*

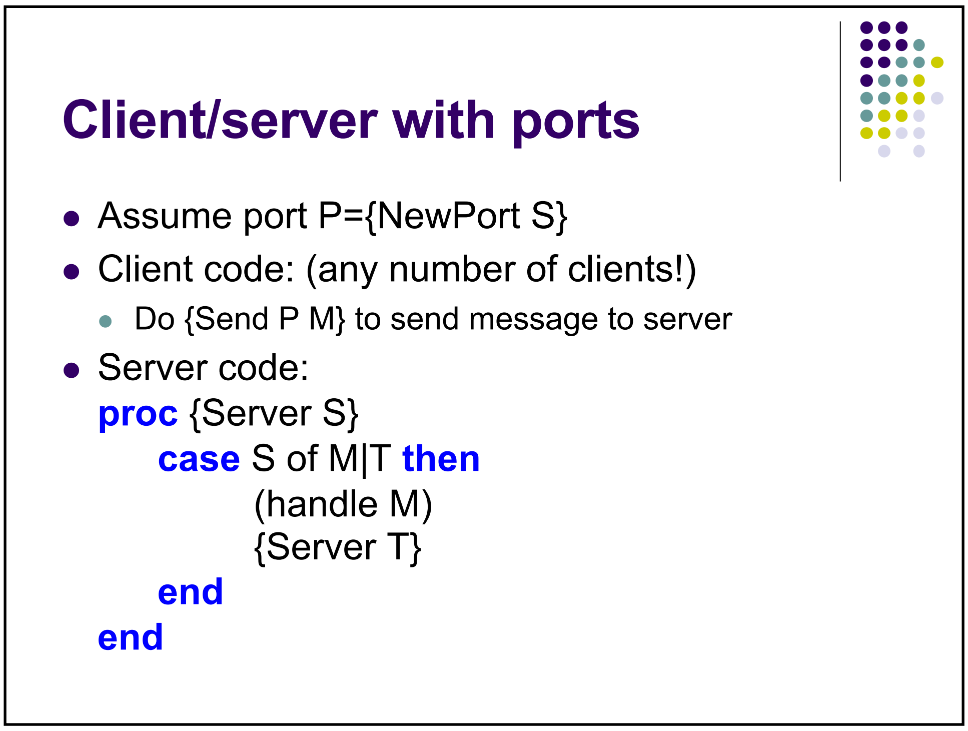
19

Port semantics (2)



- {Send X P} adds x to the end of the port's stream and updates the new end of stream
  - The send operation is atomic, which means the scheduler is guaranteed never to stop in the middle, so it happens as if it is one indivisible step
- We assume that environment *E*={P→*p*,X→*x*}

20

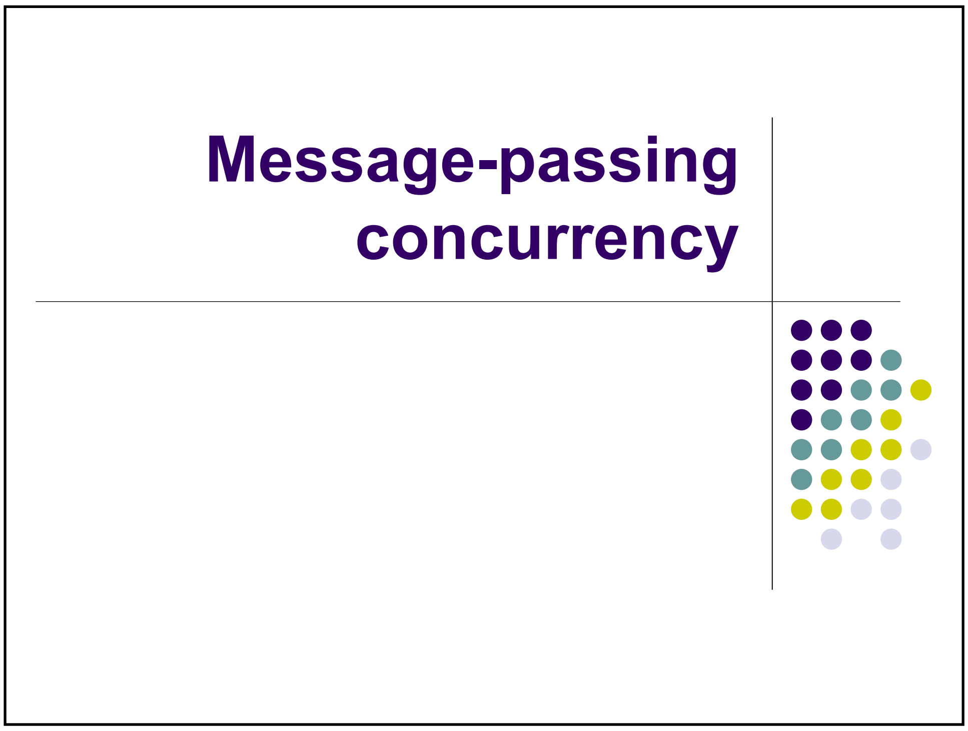


Client/server with ports

- Assume port P={NewPort S}
- Client code: (any number of clients!)
  - Do {Send P M} to send message to server
- Server code:

```
proc {Server S}
    case S of M|T then
        (handle M)
        {Server T}
    end
end
```

21



Message-passing concurrency

22

Message-passing concurrency

- Message-passing concurrency is a new paradigm for concurrent programming
  - It consists of deterministic dataflow with ports
  - It is also called multi-agent actor programming
- We show how to write concurrent programs in this new paradigm
  - We define port objects and active objects
  - We show how to do message protocols
- We then define another paradigm, deterministic dataflow with ports, which is the best all-round paradigm for concurrent programming (as far as we know)

23

Stateless port objects (stateless agents)

24

Stateless port objects

- A stateless port object is a combination of a port, a thread, and a recursive list function
  - We also call it a stateless agent
- Each agent is defined in terms of how it replies to messages
- Each agent has its own thread, so there are no problems with concurrency
- Agents are a very useful concept!

25

A math agent

- Here is a simple procedure to do arithmetic:

```
proc {Math M}
    case M
    of add(N M A) then A=N+M
    [] mul(N M A) then A=N*M
    …
    end
end
```

26

Making it a port object

- We add a port, a thread, and a recursive procedure:

```
MP={NewPort S}
proc {MathProcess Ms}
      case Ms of M|Mr then
               {Math M} {MathProcess Mr}
      end
end
thread {MathProcess S} end
```

27

Using ForAll

- We replace MathProcess by ForAll:

```
proc {ForAll Xs P}
      case Xs of nil then skip
      [] X|Xr then {P X} {ForAll Xr P}
      end
end
```

- Using ForAll, we get:

```
proc {MathProcess Ms} {ForAll Ms Math} end
```

28

Defining new port objects (1)

- A generic way to build stateless port objects:

```
fun {NewPortObject0 Process}
     Port Stream
in
     Port={NewPort Stream}
     thread {ForAll Stream Process} end
     Port
end
```

29

Defining new port objects (2)

- A generic way to build stateless port objects:

```
fun {NewPortObject0 Process}
     Port Stream
in
     Port={NewPort Stream}
     thread for M in Stream do {Process M} end end
     Port
end
```

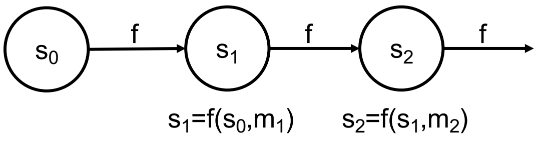
Using syntax of **for** loops

30

Stateful port objects (stateful agents)

31

Stateful port objects (Section 5.2)



$$\begin{tikzcd} s_0 \arrow[r, "f"] & s_1 \arrow[r, "f"] & s_2 \arrow[r, "f"] & {} \end{tikzcd}$$

$s_1=f(s_0,m_1)$ $s_2=f(s_1,m_2)$

- A stateful port object, also called stateful agent, has an internal memory s_i called its state
- The state is updated with each message received, which gives a state transition function:

  $F: \text{State} \times \text{Msg} \mapsto \text{State}$

32

Creating stateful port objects

- We define a generic function for stateful port objects:

```
fun {NewPortObject Init F}
    proc {Loop S State}
       case S of M|T then {Loop T {F State M}} end
    end
    P
in
    thread S in P={NewPort S} {Loop S Init} end
    P
end
```

33

Structure of Loop

- Does the Loop function ring a bell?

```
proc {Loop S State}
    case S of M|T then {Loop T {F State M}} end
end
```

- Loop starts from an initial state
- Loop successively applies F to the previous state and a message
- The function F is a binary operation
- …

34

Structure of Loop

- Does the Loop function ring a bell?

```
proc {Loop S State}
     case S of M|T then {Loop T {F State M}} end
end
```

- Loop starts from an initial state
- Loop successively applies F to the previous state and a message
- The function F is a binary operation
- Of course! It is a Fold operation!

35

FoldL operation

- FoldL is an important higher-order operation:

```
fun {FoldL S F U}
     case S
     of nil then U
     [] H|T then {FoldL T F {F U H}}
     end
end
```

36

Fold is the heart of the agent

- We replace:
  `thread S in P={NewPort S} {Loop S Init} end`
- by:
  `thread S in P={NewPort S} {FoldL S F Init} end`

- Oops! There is a small bug…

37

Updated NewPortObject

- We define a generic function for stateful port objects:

```
fun {NewPortObject Init F}
     P Out
in
     thread S in P={NewPort S} Out={FoldL S F Init} end
     P
end
```

- Out is the final state when the agent terminates
  - It never terminates here, but in another definition it might

38

Example Cell agent

- This agent behaves like a cell!

```
fun {CellProcess S M}
     case M
     of assign(New) then New
     [] access(Old) then Old=S S
     end
end
```

- Cells and ports are equivalent in expressiveness
  - Even though they look very different

39

Uniform interfaces (1)

- We can create and use a cell agent:

```
declare Cell
Cell={NewPortObject CellProcess 0}
{Send Cell assign(1)}
local X in {Send Cell access(X)} {Browse X} end
```

- We want to have the same interface as objects:

```
{Cell assign(1)}
local X in {Cell access(X)} {Browse X} end
```

40

Uniform interfaces (2)

- We change the output to be a procedure:

```
fun {NewPortObject Init F}
    P Out
in
    thread S in P={NewPort S} Out={FoldL S F Init} end
    proc {$ M} {Send P M} end
end
```

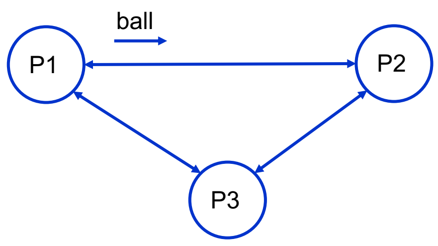
- P is hidden inside the procedure by lexical scoping
- This makes it easier to use port objects or standard objects as we saw before

41

Play ball example

42

Play ball example



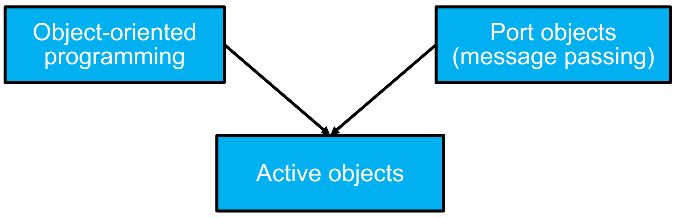
- This is a simple multi-agent program using stateful port objects
  - Three players stand in a circle. There is one ball. A player who receives the ball will send it to one of the other two, chosen randomly.
  - Each player counts the number of times it has received the ball, and it responds to a query asking for this count
- See the live lecture for the code!

43

Active objects

44

Active objects (Section 7.8)



- An active object is a port object whose behavior is defined by a class
- Active objects combine the abilities of object-oriented programming (including polymorphism and inheritance) and message-passing concurrency
- To explain active objects, we refresh your memory on object-oriented programming and we introduce classes in Oz

45

Classes and objects in Oz

- We saw objects in the course
- We now complete this explanation by introducing classes and their Oz syntax

```
class Counter
     attr i
     meth init(X)
        i := X
     end
     meth inc(X)
        i := @i + X
     end
     meth get(X)
        X=@i
     end
end
```

- Create an object:

```
Ctr={New Counter init(0)}
```

- Call the object:

```
{Ctr inc(10)}
{Ctr inc(5)}
local X in
     {Ctr get(X)}
     {Browse X}
end
```

46

Defining active objects

- Active objects are defined by combining classes and port objects
- We use the uniform interface to make them look like standard Oz objects

```
fun {NewActive Class Init}
   Obj={New Class Init}
   P
in
   thread S in
      {NewPort S P}
      for M in S do {Obj M} end
   end
   proc {$ M} {Send P M} end
end
```

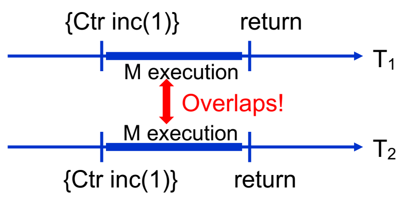
47

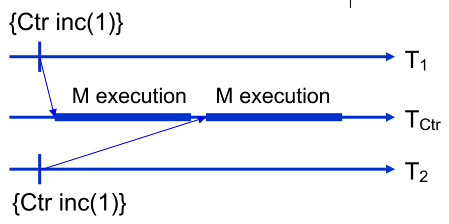
Passive objects and active objects

- We make a distinction between passive objects and active objects
- Standard objects in Oz (and many other languages, such as Java and Python) are now called passive objects
  - This is because they execute in the thread of their caller; they do not have their own thread
- This is in contrast to active objects, which have their own thread
- Let us compare passive and active objects!

48

Concurrency comparison





- Passive objects cannot be safely called from more than one thread
- The method executions can overlap, which leads to concurrency bugs

- Active objects are completely safe when called from more than one thread
- The method executions are executed sequentially in the active object's own thread

Passive objects are not concurrency-safe!

- The following code is buggy:

```
Ctr={New Counter init(0)}
thread {Ctr inc(1)} end
thread {Ctr inc(1)} end
local X in
        {Ctr get(X)}
        {Browse X}
end
```

- This can display 1! Why?
  - Look at the instruction i := @i +1
  - If the scheduler puts T1 to sleep after @i and before i:=, executes T2 fully, and then resumes T1

- The following code is correct:

```
Ctr={NewActive Counter init(0)}
thread {Ctr inc(1)} end
thread {Ctr inc(1)} end
local X in
        {Ctr get(X)}
        {Browse X}
end
```

- This will always display 2
  - Because the two methods are executed sequentially by Ctr's thread

Message protocols

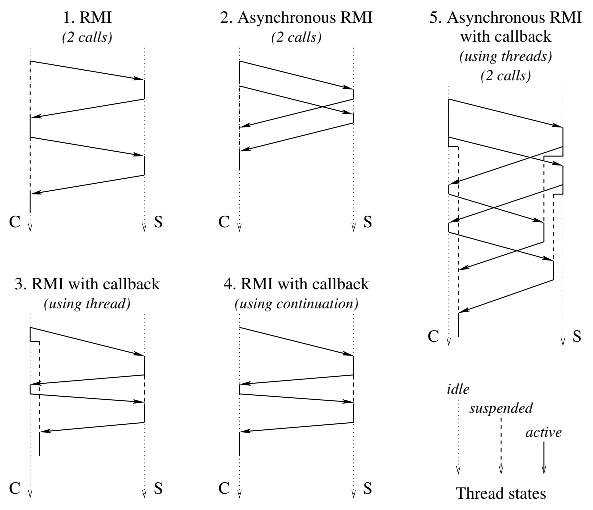
51

Message protocols (1)

- A message protocol is a sequence of messages between two or more parties that can be understood at a higher level of abstraction than individual messages
- Using port objects, let us investigate some important message protocols
- We will see the protocols using examples that are coded live
  - Explained in Section 5.3 of the course textbook

52

Message protocols (2)



- We start with a simple RMI
- We then make it asynchronous and add callbacks
- The most complicated protocol is asynchronous RMI with callback

53

Memory management and garbage collection

54

Memory management

- We give another example of using the semantics to understand program execution
  - We will explain the concept of memory management using the abstract machine
  - Memory management is important for all paradigms
- Managing program memory during execution
  - We have already explained the advantages of last call optimization using the abstract machine
  - Now we will explain automatic memory management, which is also known as garbage collection

55

Example of memory management

- Consider the following simple program:

```
proc {Loop10 I}
    if I==10 then skip
        {Browse I}
        {Loop10 I+1}
    end
end
```

- Calling {Loop10 0} displays integers 0 up to 9

56

Execution of {Loop10 0} (1)

- We show part of the execution states:
  $([(\{\text{Loop10 0}\},E_0)], \sigma) \rightarrow$
  $([(\{\text{Browse I}\},\{I\rightarrow i_0\}),(\{\text{Loop10 I+1}\},\{I\rightarrow i_0\})], \sigma\cup\{i_0=0\}) \rightarrow$
  $([(\{\text{Loop10 I+1}\},\{I\rightarrow i_0\})], \sigma\cup\{i_0=0\}) \rightarrow$
  $([(\{\text{Browse I}\},\{I\rightarrow i_1\}),(\{\text{Loop10 I+1}\},\{I\rightarrow i_1\})], \sigma\cup\{i_0=0,i_1=1\}) \rightarrow$
  $([(\{\text{Loop10 I+1}\},\{I\rightarrow i_1\})], \sigma\cup\{i_0=0,i_1=0\}) \rightarrow$
  …
  $([(\{\text{Browse I}\},\{I\rightarrow i_9\}),(\{\text{Loop10 I+1}\},\{I\rightarrow i_9\})], \sigma\cup\{\underbrace{i_0=0,i_1=1, \ldots,}_{\boxed{\substack{\{i_0,\ldots,i_8\}\text{ not needed!}\\ \text{Only } i_9 \text{ is needed (from stack)}}}} i_9=9\}) \rightarrow$
  …
- You can observe two things:
  - The stack size is constant (last call optimization)
  - The memory size keeps growing
    - But most of the variables are only used briefly and then no longer needed!
  - We will see how to remove the unneeded variables

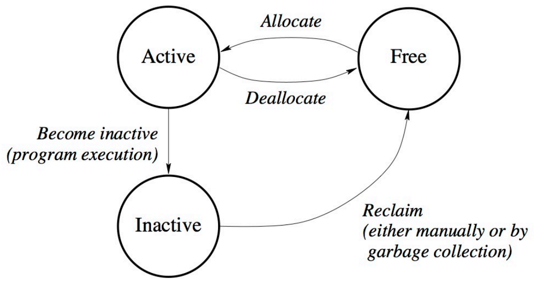
57

Data representation in memory

- Programs execute in main memory, which consists of a set of memory words
  - In today's personal computers, a memory word has a 64-bit size (in older computers they have 32-bit size)
  - Memory words are used to represent the semantic stack and the memory (variables and cells)
- When the operating system starts a process, it gives the process an initial set of memory words for its execution
  - This set can be increased or reduced in size by system calls
  - During program execution, the set of words is managed by the process, i.e., by the executing program

58

Life cycle of a memory word



- A memory word can have three states: active, inactive, and free
- When execution starts, all words are in the free pool
- When the program needs a word, it allocates one from the free pool
- It may happen that a program no longer needs a word:
  - If it knows that it no longer needs it, it deallocates it (puts it back on the free pool)
  - If it does not know, then the word becomes inactive (a kind of limbo state!)

59

When reclaiming goes wrong

- Inactive memory blocks must eventually be put back in the free pool ("reclaimed")
- Two kinds of problems can occur if this is done wrong:
  - Dangling reference: when a word is reclaimed even though it is still reachable. The system thinks the word is inactive, but it is still active.
  - Memory leak: when a word is never reclaimed even though it is not reachable. Too many memory leaks can cause the process to crash or exhaust the computer's resources.
- Reclaiming can be done manually (by the programmer) or by an algorithm (garbage collection)
  - Manual reclaiming is quite tricky and not recommended!
  - Garbage collection is much more reliable, if the garbage collection algorithm is correctly implemented. We will see how this works!

60

Execution of {Loop10 0} (2)

- We show part of the execution states:
  ([({Loop10 0},E_0)], σ) →
  ([({Browse I},{I→i_0}),({Loop10 I+1},{I→i_0})], σ∪{i_0=0}) →
  ([({Loop10 I+1},{I→i_0})], σ∪{i_0=0}) →
  ([({Browse I},{I→i_1}),({Loop10 I+1},{I→i_1})], σ∪{i_0=0,i_1=1}) →
  ([({Loop10 I+1},{I→i_1})], σ∪{i_0=0,i_1=0}) →
  …
  ([({Browse I},{I→i_9}),({Loop10 I+1},{I→i_9})], σ∪{i_0=0,i_1=1, …, i_9=9}) →
  …
- We can observe the life cycle of these words:
  - When the stack shrinks, its words can be immediately deallocated (become free)
  - The memory $σ_1$ never shrinks: we only add new variables
    - The same thing happens to the cell store $σ_2$: we only add new cells
  - All unneeded variables and cells become inactive: can we find out which ones?

61

When is a word inactive?

- A word becomes inactive when it is unreachable from the stack
- Consider an example execution state:
  ([({Browse I},{I→i_9}),({Loop10 I+1},{I→i_9})], σ∪{i_0=0,i_1=1, …, i_9=9})
- The words used to represent variables i_0 up to i_8 are inactive
- Consider another example execution state:
  ([({Browse I},{I→k_1})], {k_2=b|k_1, k_1=c|k_0, b=5, c=6, k_0=nil})
- The browse refers to list 6|nil stored in memory (variables k_1, c, k_0)
  - The stack contains k_1, which refers indirectly to c and k_0. They are all reachable!
- The words used to represent variables k_2 and b are inactive
- Is there a way to detect the inactive variables and make them free?

62

Finding inactive words

- Program execution is determined by the stack
  - All words used to represent variables and cells reachable from the stack (directly or indirectly) are active
- To make words free, we need to find the variables and cells that are unreachable from the stack
  - This is not a simple algorithm
  - Technically, the algorithm does transitive closure of the one-step reachability relation
  - We give a formal definition of reachability
- All variables and cells that are reachable are active
  - All the others (unreachable!) are inactive and can be made free

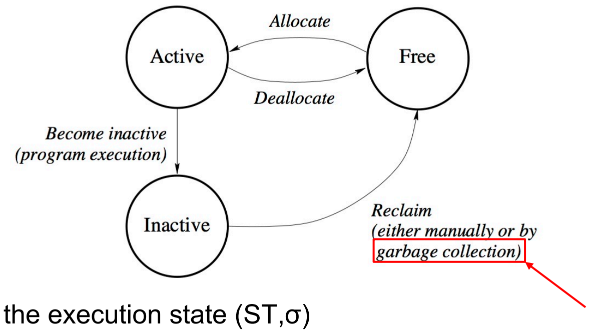
63

Reachability relation

- We define an algorithm to compute the reachability of all variables
  - Given an execution state (ST,σ) where ST is the stack and $\sigma=\sigma_1 \cup \sigma_2$ where σ_1 is the variable store and σ_2 is the cell store
  - Denote by V_σ and V_{ST} the set of all variables in σ and ST, respectively
- We first define the one-step reachability relation as follows:
  - $x \longmapsto y$: if $x=r(\ldots f_i{:}y \ldots) \in \sigma_1$ (record in the variable store)
  - $x \longmapsto y$: if $x{:}y \in \sigma_2$ (cell in the cell store)
- We say that a variable $x \in V$ is reachable if there exists a path $x_0 \longmapsto x_1 \longmapsto x_2 \longmapsto \ldots \longmapsto x_{n-1} \longmapsto x$ (of any length ≥ 0) such that:
  - $x_0 \in V_{ST}$
  - $1 \leq \forall i < n : x_{i-1} \longmapsto x_i$ holds
- All reachable variables are represented by active words. All variables that are not reachable are represented by inactive words and can be made free.

64

Garbage collection



- Given the execution state (ST,σ)
  - A word becomes inactive when it is directly or indirectly unreachable from the stack
  - An executing program does not immediately know when a word becomes inactive
- Garbage collection determines which variables and cells are inactive
  - All words used to represent inactive variables and cells can be made free
  - The algorithm is complex and time-consuming: deciding when to execute the algorithm is a difficult task, because it should not hinder normal execution while at the same time allow efficient memory management

Execution of {Loop10 0} (3)

- We show part of the execution states:
  ([({Loop10 0},E_0)], σ) →
  ([({Browse I},{I→i_0}),({Loop10 I+1},{I→i_0})], σ∪{i_0=0}) →
  ([({Loop10 I+1},{I→i_0})], σ∪{i_0=0}) →
  ([({Browse I},{I→i_1}),({Loop10 I+1},{I→i_1})], σ∪{i_0=0,i_1=1}) →
  ([({Loop10 I+1},{I→i_1})], σ∪{i_0=0,i_1=0}) →
  …
  ([({Browse I},{I→i_9}),({Loop10 I+1},{I→i_9})], σ∪{i_0=0,i_1=1, …, i_9=9})
  $\rightarrow_{GC}$
  ([({Browse I},{I→i_9}),({Loop10 I+1},{I→i_9})], σ∪{i_9=9})
- Let us run garbage collection on the final execution state
  - The garbage collection algorithm does (ST,σ) $\rightarrow_{GC}$ (ST',σ')
  - Garbage collection removes variables {i_0, …, i_8} and their bindings
  - Garbage collection has no influence on program execution: the program will give the same results with or without garbage collection.

Garbage collection today

- Many modern programming languages do garbage collection
  - Java, Python, Erlang, Oz, Haskell, Scheme, …
- A few low-level languages that allow direct access to the processor do not do garbage collection
  - Assembly language, C, C++
  - For some low-level tasks, such as writing device drivers, it is important to do only manual memory management
  - Even so, there exist "conservative GC" algorithms for C and C++

67

Active memory versus memory consumption

Garbage collection

- Active memory is how many words the program needs at any time
  - An in-memory database has a large active memory (= the size of the database) but a small memory consumption (= little memory is needed to calculate the result of a query)
- Memory consumption is the number of words allocated per time unit
  - A simulation of molecules moving in a box has a large memory consumption (= each particle position is recalculated at every time step according to a complex computation that needs much temporary data) but a small active memory (= little memory is needed to store positions and velocities of all particles)

Intuition: Your active size is how much you weigh (in kg); your food consumption is how much you eat (in kg/day)

- The food you eat is used by your metabolism but only a small part (or none) becomes part of your body! Even if you eat 2 kg/day you won't weigh 200 kg after 100 days.

68

Deterministic dataflow with ports

69

The best way (as far as I know)

- Writing general concurrent programs is difficult!
  - But deterministic dataflow is easy ("Concurrency for Dummies")
  - Can this paradigm offer help for general programs? Yes!
- This leads to the best way to write concurrent programs
  - Start with deterministic dataflow as the default
  - Add ports where they are needed, but as few as possible
  - This differs from message passing (multi-agent actors) in that we don't use port objects or active objects directly
- We give some example designs using this approach
  - Concurrent composition (static and dynamic)
  - Eliminating sequential dependencies

70

Concurrent composition (fixed number of threads)

71

Concurrent composition (Section 4.4.3)

- The **thread** statement creates a thread that executes independently of the original thread

```
thread <s>1 end
thread <s>2 end
% Two new threads with <s>1 and <s>2, original thread continues
```

- Sometimes the new threads have to be subordinate to the original
  - The original thread waits until the new threads have terminated
- This operation is called concurrent composition

```
(<s>1 || <s>2) % Create two threads and wait until both are terminated
<s>3           % Executes only after both are done
```

72

Implementation

- We implement ($<s>_1$ || $<s>_2$) using dataflow variables
  - We use the constant **unit** when the value does not matter

```
local X1 X2 in
   thread <s>1 X1=unit end
   thread <s>2 X2=unit end
   {Wait X1}
   {Wait X2}
end
```

- It does not matter in which order we wait

73

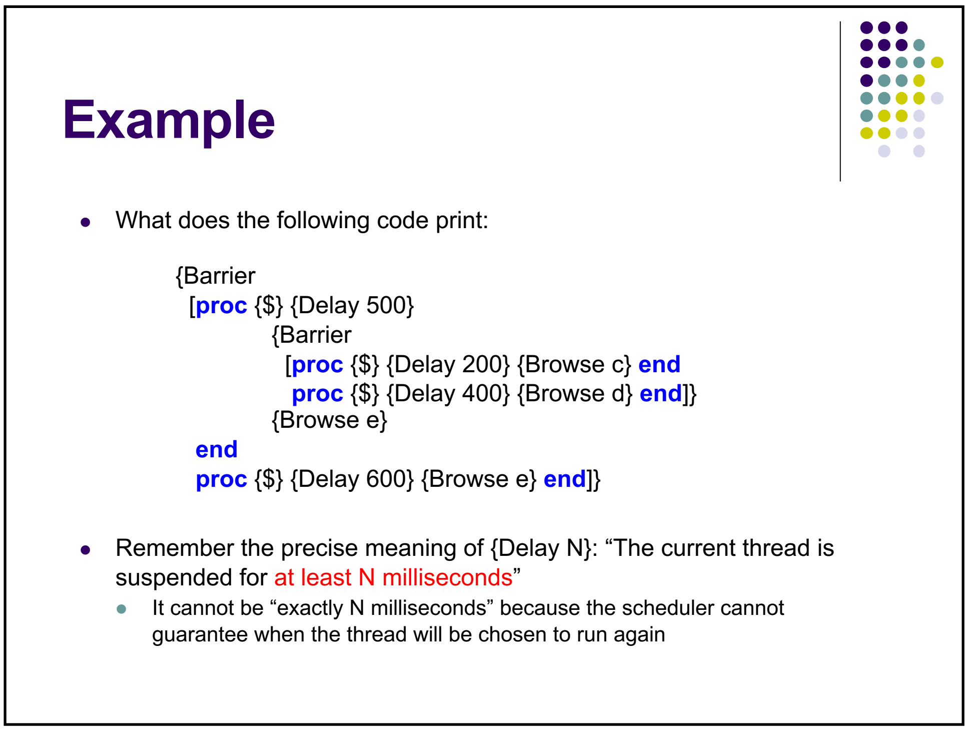
Higher-order abstraction

- Using higher-order programming, we implement the general form:
  ($<s>_1$ || $<s>_2$ || … || $<s>_n$)
- The instruction $<s>_1$ is written as **proc** {$} $<s>_1$ **end**
- We define the procedure {Barrier Ps} with list of statements Ps:

```
proc {Barrier Ps}
   Xs={Map Ps fun {$ P} X in thread {P} X=unit end X end}
in
   for X in Xs do {Wait X} end
end
```

- Note that Barrier can be defined using deterministic dataflow only
  - No ports needed; we will add one port later when we make it dynamic

74



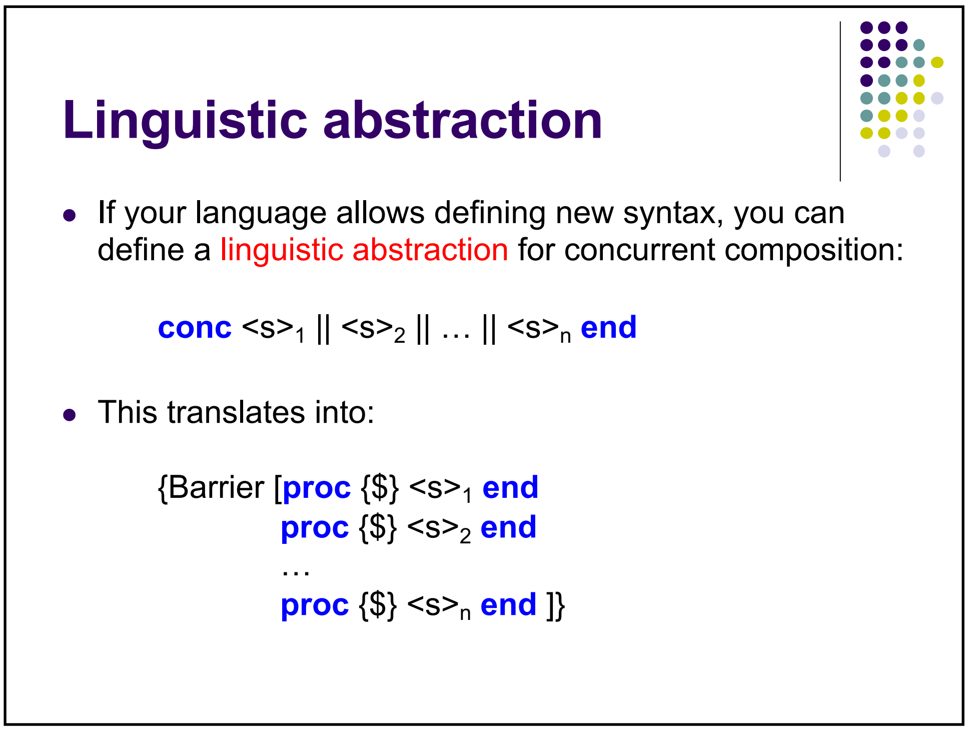
Example

- What does the following code print:

```
{Barrier
 [proc {$} {Delay 500}
            {Barrier
             [proc {$} {Delay 200} {Browse c} end
              proc {$} {Delay 400} {Browse d} end]}
            {Browse e}
  end
  proc {$} {Delay 600} {Browse e} end]}
```

- Remember the precise meaning of {Delay N}: “The current thread is suspended for at least N milliseconds”
  - It cannot be “exactly N milliseconds” because the scheduler cannot guarantee when the thread will be chosen to run again

75



Linguistic abstraction

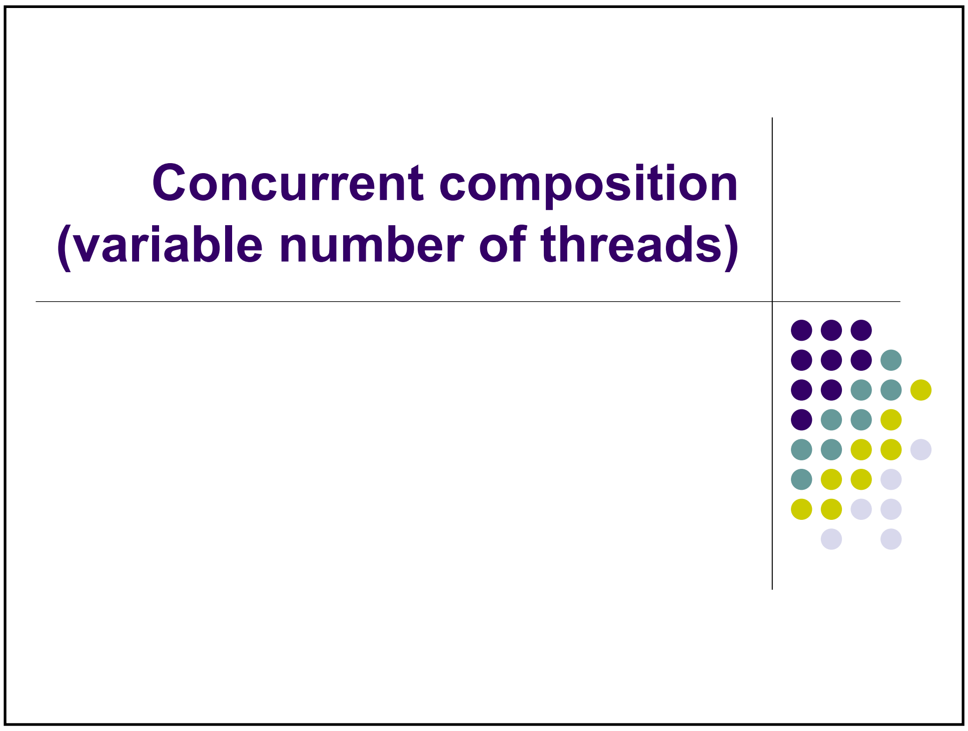
- If your language allows defining new syntax, you can define a linguistic abstraction for concurrent composition:

  conc <s>$_1$ || <s>$_2$ || … || <s>$_n$ **end**

- This translates into:

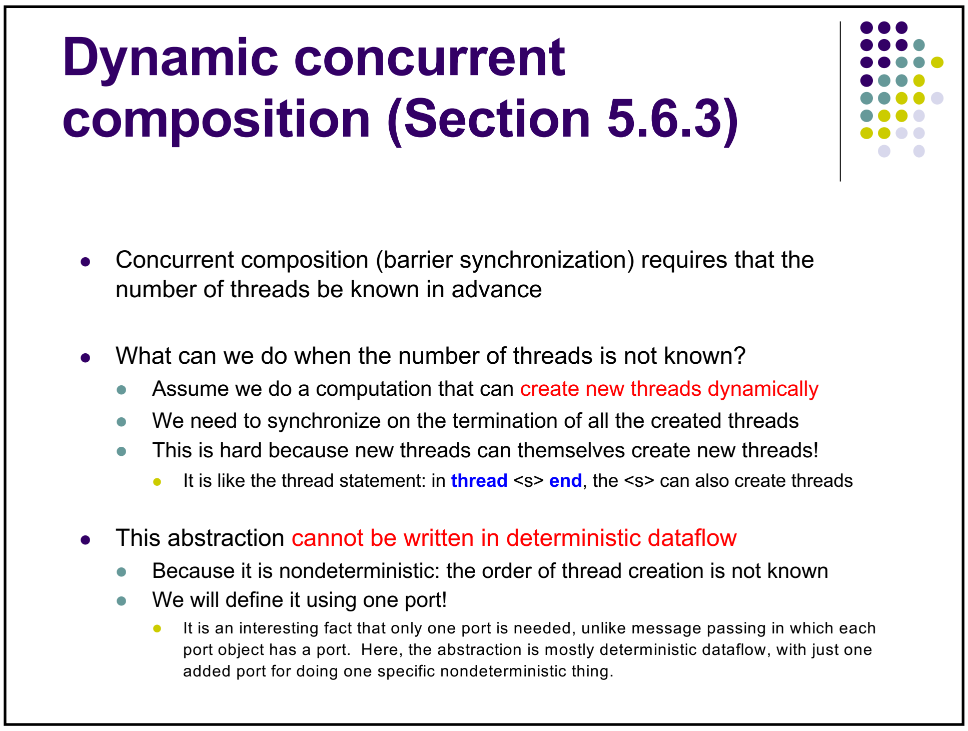
  {Barrier [**proc** {\$} <s>$_1$ **end**
  proc {\$} <s>$_2$ **end**
  …
  proc {\$} <s>$_n$ **end**]}

76



Concurrent composition (variable number of threads)

77



Dynamic concurrent composition (Section 5.6.3)

- Concurrent composition (barrier synchronization) requires that the number of threads be known in advance
- What can we do when the number of threads is not known?
  - Assume we do a computation that can create new threads dynamically
  - We need to synchronize on the termination of all the created threads
  - This is hard because new threads can themselves create new threads!
    - It is like the thread statement: in **thread** <s> **end**, the <s> can also create threads
- This abstraction cannot be written in deterministic dataflow
  - Because it is nondeterministic: the order of thread creation is not known
  - We will define it using one port!
    - It is an interesting fact that only one port is needed, unlike message passing in which each port object has a port. Here, the abstraction is mostly deterministic dataflow, with just one added port for doing one specific nondeterministic thing.

78

Specifying the abstraction

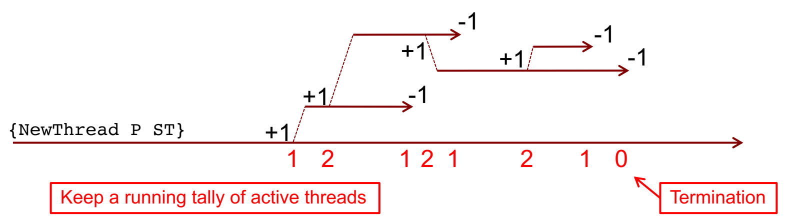
- The main thread waits until all subordinate threads are terminated
- We can define this abstraction as follows:

```
{NewThread proc {$} <s> end SubThread}
{SubThread proc {$} <s> end}
```

- NewThread creates a new computation with <s> in the main thread and outputs the procedure SubThread
  - NewThread terminates only after all subordinate threads are terminated
- SubThread creates a subordinate thread with <s>
  - Both <s> are allowed to call SubThread, and so on recursively, so the tree of threads can be arbitrarily deep

79

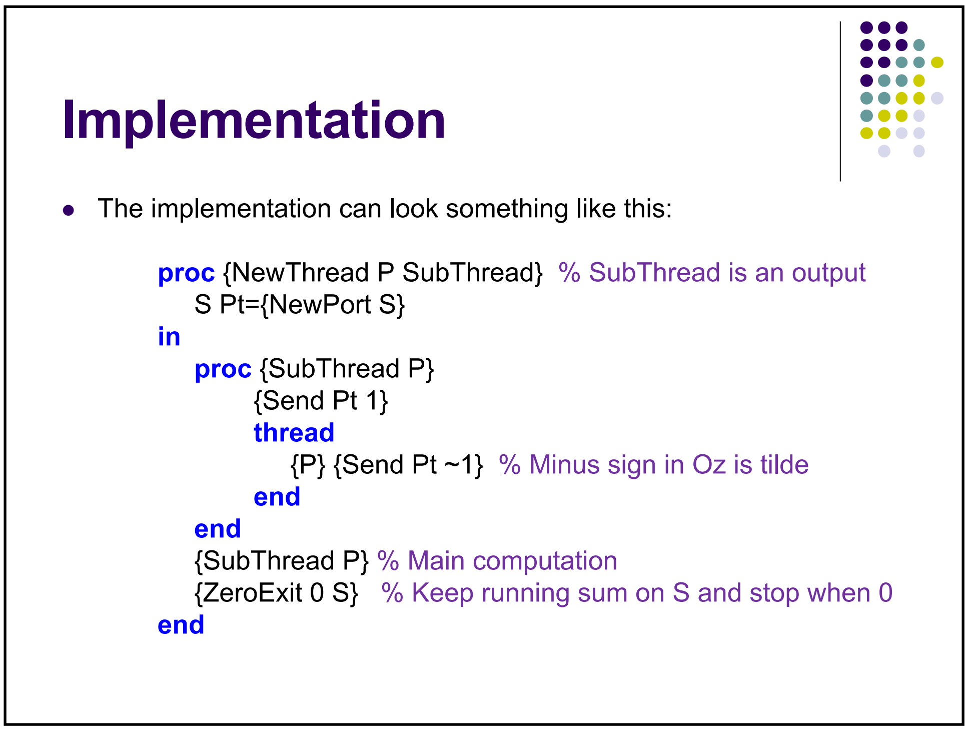
Algorithm



- We use a port to count the number of active threads
- Each new thread sends +1 to the port when it is created and -1 to the port when it terminates
  - This is trickier than it seems: send +1 *just before* creation and -1 *inside the thread* just before termination (we need to make a proof)
  - When the running total on the stream is 0 then all threads are terminated

80

80

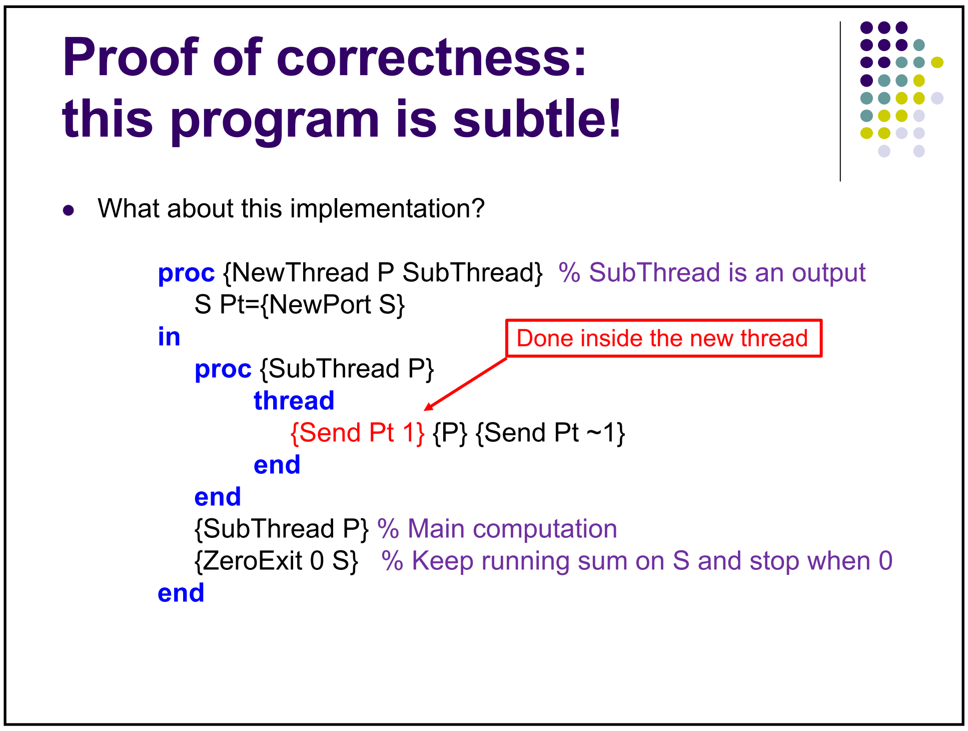


Implementation

- The implementation can look something like this:

```
proc {NewThread P SubThread}  % SubThread is an output
   S Pt={NewPort S}
in
   proc {SubThread P}
        {Send Pt 1}
        thread
           {P} {Send Pt ~1}  % Minus sign in Oz is tilde
        end
   end
   {SubThread P} % Main computation
   {ZeroExit 0 S}   % Keep running sum on S and stop when 0
end
```

81



Proof of correctness: this program is subtle!

- What about this implementation?

```
proc {NewThread P SubThread}  % SubThread is an output
   S Pt={NewPort S}
in                                      Done inside the new thread
   proc {SubThread P}
        thread
           {Send Pt 1} {P} {Send Pt ~1}
        end
   end
   {SubThread P} % Main computation
   {ZeroExit 0 S}   % Keep running sum on S and stop when 0
end
```

82

Proof of correctness: buggy version!

- What about this implementation? It is buggy! Do you see why?

```
proc {NewThread P SubThread}
   S Pt={NewPort S}
in
   proc {SubThread P}
        thread
           {Send Pt 1} {P} {Send Pt ~1}
        end
   end
   {SubThread P} % Main computation
   {ZeroExit 0 S}  % Keep running sum on S and stop when 0
end
```

We need a proof!

83

Proof of correctness: invariant assertion

- We can prove correctness by using an invariant assertion
- Consider the following assertion:
  - (the sum of the elements on S) ≥ (the number of active threads)
  - When the sum is zero, it implies the number of active threads is zero
- We use induction on execution steps to show that this is always true
  - **Base case**: True at the call to NewThread since both numbers are zero
  - **Inductive case**: there are four relevant actions (see next slide!)
- The invariant assertion is just a safety property, what about liveness?
  - The first call to SubThread sends 1 to S, so we have to wait until the first created thread terminates

84

Inductive case

- During any execution, there are four possible execution steps that can change the truth of the assertion:
  - Sending 1 : clearly keeps the assertion true
  - Starting a thread : keeps the assertion true since it follows a send of 1, and the assertion was true just before the send
  - Sending ~1 : we can assume without loss of generality that thread termination occurs just before sending ~1, since the thread no longer does any work after the send
  - Terminating a thread : clearly keeps the assertion true
- You see why the {Send Pt 1} must be done outside of the new thread!
  - {Send Pt 1} must be done before creating the new thread

85

ZeroExit procedure

- The procedure {ZeroExit N S} keeps a running sum of elements from S and exits when the sum equals 0

Always read at least one element

```
proc {ZeroExit N S}
   case S of X|S2 then
      if N+X==0 then skip
      else {ZeroExit N+X S2} end
   end
end
```

86

Eliminating sequential dependencies

87

Eliminating sequential dependencies (Section 5.6.4)

- A sequential program orders all instructions
  - This is a sequential dependency, by definition!
- But sometimes these dependencies are useless and may cause the program to block unnecessary
  - Can we get rid of these dependencies?
- The solution is to add threads to remove useless dependencies, but without changing the result
  - In deterministic dataflow, we can add threads wherever we want, if the computation in the thread is purely functional
  - In our example, we will need one port to collect the elements computed in each thread: this adds nondeterminism only in one place, so we can easily check that it is ok

88

Example: The Filter function

- The function {Filter L F} takes a list L and a one-argument boolean function F and outputs the list of elements where the function is true:

```
fun {Filter L F}
   case L of nil then nil
   [] X|L2 then
      if {F X} then X|{Filter L2 F} else {Filter L2 F} end
   end
end
```

- This is efficient, but it introduces sequential dependencies! The call:
  `{Filter [A 5 1 B 4 0 6] fun {$ X} X>2 end}`
  blocks right away on A, even though we know that 5, 4, and 6 will eventually be in the output. Waiting for A stops everything!

89

Filter without sequential dependencies

- Let us write a new version of Filter that avoids these dependencies
  - It will construct its output incrementally, as input information arrives
- We can write ConcFilter using two building blocks:
  - Concurrent composition (as seen before): {Barrier Ps}
  - Asynchronous channel (port with a Close operation)
- ConcFilter removes dependencies but is nondeterministic:
  `{ConcFilter [A 5 1 B 4 0 6] fun {$ X} X>2 end}`
  - This returns right away with 5|4|6|… and will eliminate 1 and 0
  - But it can return the elements in any order, 6|5|4|… for example
- We have traded off dependencies for nondeterminism

90

Ports with a Close operation

- We need a port that can be closed (ending the stream with nil)
- We define {NewPortClose S Send Close}
  - S is the port's stream
  - {Send M} sends message M to the port
  - {Close} closes the port, i.e., binds the tail to nil and no more send is allowed
- Definition: (defined with a cell!)

```
proc {NewPortClose S Send Close}
   PC={NewCell S}
in
   proc {Send M} S in {Exchange PC  M|S  S} end
   proc {Close} nil=@PC end
end
```

- The cell PC is like an object attribute: it allows reading and writing
  - The Exchange operation does both read and write atomically
  - Exchange is needed to make the Send concurrency-safe (see passive objects!)

91

ConcFilter idea

- The original {Filter L F} computes all {F X} in the same thread
- The new {ConcFilter L F} computes each {F X} in a separate thread
  - If {F X} returns true, then send X to the port
  - The port's stream is the function's output
- When all threads terminate, the port is closed
  - This makes the stream into a list
  - We use {Barrier Ps} to detect when all threads terminate
- Creating the procedure arguments to Barrier
  - For each X in L, we need to execute **if** {F X} **then** {Send X} **end**
  - So we create the procedure **proc** {$} **if** {F X} **then** {Send X} **end end**

92

Defining ConcFilter

- ConcFilter uses Map to build the arguments to Barrier:

```
proc {ConcFilter L F L2}
   Send Close
in
   {NewPortClose L2 Send Close}
   {Barrier
        {Map L  % For each X of the input list, build procedure
           fun {$ X}
                proc {$} if {F X} then {Send X} end end
           end}}
   {Close}
end
```

Procedure input to Barrier

93

Conclusion

94

Conclusion: how to build concurrent programs

- We have seen three good paradigms for concurrent programs
- Deterministic dataflow (including lazy): best, but has limitations
  - Cannot express programs that need nondeterminism, like client/server
  - Is widely used in cloud analytics tools (e.g., Apache Flink, Spark)
- Message passing (multi-agent actor): fully general, but harder
  - Stateful agents that communicate with asynchronous messages
  - **Erlang** is a successful industrial example of this approach
- Deterministic dataflow with ports: best all-round approach
  - Write most of the program as deterministic dataflow
  - Add ports only where they are needed; usually very few are needed
  - This is a novel approach that will likely appear more in the future

We will see Erlang next week!

95

Introduction to Erlang

96

Introduction to Erlang (Section 5.7)

- The Erlang language was originally developed by Ericsson for telecommunications applications in 1986 (Java was developed in 1991)
  - It is released as OTP (Open Telecom Platform) with a full set of libraries
  - It is supported by Ericsson, the Erlang Ecosystem Foundation, and a large user community (www.erlang.org)
- Erlang programs consist of "processes", which are port objects and communicate using asynchronous FIFO message passing
  - Erlang processes share nothing: all data is copied between them
  - Erlang processes receive messages through a mailbox that is accessed by pattern matching. Messages can be received out of order if they match.
- Erlang supports building reliable long-lived distributed systems
  - Successful "let it crash" philosophy using failure linking and supervisor trees
  - Ericsson AXD 301 ATM switch with 1.7 million lines of Erlang claims 99.9999999% availability (one may doubt the number of 9's, but the system is extremely available!)