

LINFO1104 – LSINC1104

Concepts, paradigms, and semantics of programming languages

Lecture 8-9

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

© 2022 P. Van Roy. All rights reserved.



1

Overview of lectures 8-9

- Exceptions
 - How to handle exceptional situations in a program without making the program text more complicated
- Concurrent programming
 - Deterministic dataflow (a.k.a. functional dataflow)
 - Semantics of concurrent programming
 - “Concurrency for Dummies”
 - Programming techniques for deterministic dataflow

© 2022 P. Van Roy. All rights reserved.



2

Exceptions



© 2022 P. Van Roy. All rights reserved.

3

How to handle exceptional situations



- How can we handle exceptional situations in a program?
 - Such as: division by 0, opening a nonexistent file, and so forth
 - Program errors but also errors from outside the program
 - Things that happen rarely but that must be taken care of
- We add a **new programming concept** called **exceptions**
 - We define exceptions and show how they are used
 - We give the semantics of exceptions in the abstract machine
- With exceptions, we can handle exceptional situations without cluttering up the program with rarely used error checking code

© 2022 P. Van Roy. All rights reserved.

4

The containment principle

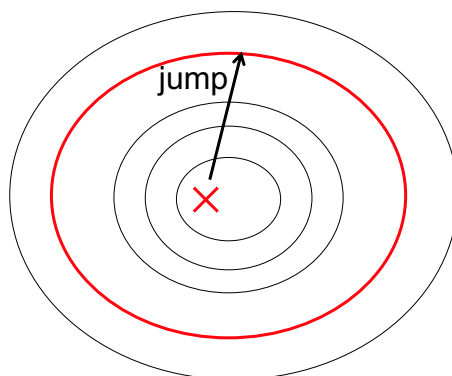


- When an error occurs, we would like to be able to recover from the error
- Furthermore, we would like the error to affect as little as possible of the program
- We propose **the containment principle**:
 - A program is a set of **nested execution contexts**
 - An error will occur **inside** an execution context
 - A recovery routine (exception handler) exists at the boundary of an execution context, to make sure the error **does not propagate** to higher execution contexts

© 2022 P. Van Roy. All rights reserved.

5

Handling an exception



✗ An error that raises an exception

○ An execution context

○ The execution context that catches the exception

- An executing program that encounters an error must jump to another part (the exception handler) and give it a reference (the exception) that describes the error

© 2022 P. Van Roy. All rights reserved.

6

The try and raise instructions



- We introduce two new instructions for handling exceptions:

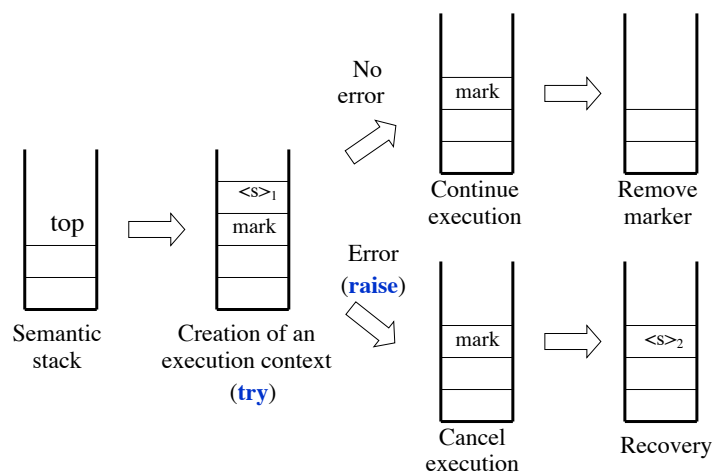
```
try <s>1 catch <y> then <s>2 end % Create an execution context
raise <x> end % Raise an exception
```

- With the following behavior:
 - try** puts a “marker” on the stack and starts executing <s>₁
 - If there is no error, <s>₁ executes normally and removes the marker when it terminates
 - raise** is executed when there is an error, which empties the stack up to the marker (the rest of <s>₁ is therefore canceled)
 - Then <s>₂ is executed
 - <y> refers to the same variable as <x>
 - The scope of <y> exactly covers <s>₂

© 2022 P. Van Roy. All rights reserved.

7

Semantics of exceptions

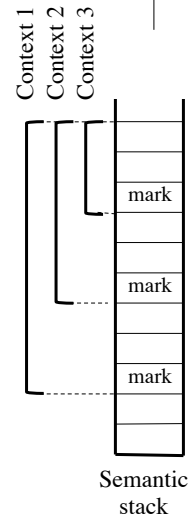
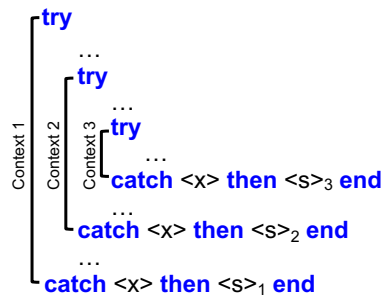


© 2022 P. Van Roy. All rights reserved.

8

An execution context

- An **execution context** is the part of the semantic stack that starts with a marker and continues to the stack top:



© 2022 P. Van Roy. All rights reserved.

9

Example using exceptions

```

fun {Eval E}
  if {IsNumber E} then E
  else
    case E
    of plus(X Y) then {Eval X}+{Eval Y}
    [] times(X Y) then {Eval X}*{Eval Y}
    else raise badExpression(E) end
    end
  end
end

try
  {Browse {Eval plus(23 times(5 5))}}
  {Browse {Eval plus(23 minus(4 3))}}
catch X then {Browse X} end
  
```

- Using exceptions, the error handling code does not clutter up the program

© 2022 P. Van Roy. All rights reserved.

10

If we did not have exceptions...



```
fun {Eval E}
  if {IsNumber E} then E
  else
    case E
    of plus(X Y) then R={Eval X} in
      case R of badExpression(RE) then badExpression(RE)
      else R2={Eval Y} in
        case R2 of badExpression(RE) then badExpression(RE)
        else R+R2
      end
    end
    [] times(X Y) then
      % ... Same code as plus
    else badExpression(E)
    end
  end
end
```

- Much more code!
 - In this example, 22 lines instead of 10 (more than double)
- The code is much more complicated because of all the **case** statements handling badExpression

© 2022 P. Van Roy. All rights reserved.

11

The “finally” clause



- The **try** has an additional **finally** clause, for an operation that must always be executed (in both the correct and error cases):

```
FH={OpenFile “foobar”}
try
  {ProcessFile FH}
catch X then
  {Show “*** Exception during execution ***”}
finally {CloseFile FH} end % Always close the file
```

© 2022 P. Van Roy. All rights reserved.

12

Exceptions in Java



- An exception is an object that inherits from the class Exception (which is a subclass of Throwable)
- There are two kinds of exceptions
 - **Checked exceptions:** The compiler verifies that all methods only throw the exceptions declared for the class
 - **Unchecked exceptions:** Some exceptions can arrive without the compiler being able to verify them. They inherit from RuntimeException and Error.
- For exceptions that the program itself defines, you should **always use checked exceptions**, since they are declared and therefore part of the program's interface

© 2022 P. Van Roy. All rights reserved.

13

Java exception syntax



```
throw new NoSuchElementException(name) ;

try {
    <stmt>
} catch (exctype1 id1) {
    <stmt>
} catch (exctype2 id2) {
    ...
} finally {
    <stmt>
}
```

© 2022 P. Van Roy. All rights reserved.

14



Good style

- We read a file and perform an action for each item in the file:

```
try
    while (!stream.eof())
        process(stream.nextTokn());
finally
    stream.close();
```

© 2022 P. Van Roy. All rights reserved.

15



Bad style

- We can use the exception handler to change the execution order **during normal execution**:

```
try {
    for (;;)
        process (stream.next());
} catch (StreamEndException e) {
    stream.close();
}
```

- Reaching the end of a stream is completely **normal**, it is not an error. What happens if a **real error** happens and is mixed in with the normal operation? You don't want to mix things. Normal operation should be kept separate from errors!

© 2022 P. Van Roy. All rights reserved.

16

Conclusions on data abstraction



© 2022 P. Van Roy. All rights reserved.

17

Java, Scala, and language design



- Java was designed to support data abstraction (1990s)
 - True data abstraction (encapsulation, GC)
 - All entities are objects or ADTs
 - Support for object-oriented design principles
- Scala has added two principles to this (2000s)
 - Strict separation between mutable/immutable
 - Everything is an object (including functions)
- These principles considerably increase Scala's expressive power compared to Java
 - We consider that Scala is an important successor to Java
 - Although some people consider it is a Swiss Army knife!

© 2022 P. Van Roy. All rights reserved.

18

Final remarks



- This completes the part of the course on data abstraction
 - Explicit state, objects, and ADTs
 - Exceptions
 - We did not go in-depth into object-oriented programming techniques because they are covered in many other courses
- So far in the course we have covered **three important themes**
 - **Functional programming** (including recursion, invariant programming, and higher-order programming)
 - **Language semantics** (including a complete operational semantics and an introduction to lambda calculus)
 - **Data abstraction** (including explicit state, objects, and ADTs)
- The next theme is **concurrent programming**

© 2022 P. Van Roy. All rights reserved.

19

Concurrent programming



© 2022 P. Van Roy. All rights reserved.

20

The world is concurrent



- The real world is **concurrent**
 - It is made of **activities that progress independently**
- The computing world is concurrent too:
 - **Distributed system**: computers linked by a network
 - A concurrent activity is called a **computing node (computer)**
 - Each computing node has its own resources (memory, CPU)
 - **Operating system**: management of a single computer
 - A concurrent activity is called a **process**
 - Processes share the same computer resources and have independent memory spaces
 - **Process**: execution of a single program
 - A concurrent activity is called a **thread**
 - Threads share the same memory space

© 2022 P. Van Roy. All rights reserved.

21

Concurrent programming



- Concurrency is natural
 - Many activities are naturally independent
 - Activities that are **independent** are ipso facto **concurrent**
 - So how can we write a program with many independent activities?
 - Concurrency must be supported by the language!
- A concurrent program
 - Multiple progressing activities that exist at the same time
 - Activities that can communicate and synchronize
 - **Synchronize**: an activity waits for an action of another activity
 - **Communicate**: information passes from one activity to another

© 2022 P. Van Roy. All rights reserved.

22

Concurrency can be (very) hard



- It introduces many difficulties such as nondeterminism, race conditions, reentrancy, deadlocks, livelocks, fairness, consistency of shared data
 - Java's **synchronized objects** are tough to program with
 - Erlang's and Scala's **actors** are better, but they still have race conditions
 - **Libraries** can hide some of these problems, but they always peek through
- Adding distribution makes it **even harder**
- Adding partial failure makes it **even much harder than that**
- The Holy Grail: can we make concurrent programming as easy as sequential programming?
 - Yes, it can be done, if the paradigm is chosen wisely
 - In this course we will see **deterministic dataflow**, which is a concurrent paradigm that is a form of functional programming

LINFO2345

© 2022 P. Van Roy. All rights reserved.

23

Deterministic dataflow



© 2022 P. Van Roy. All rights reserved.

24

Concurrency paradigms



- There are **three main paradigms** of concurrent programming
- The simplest is called **deterministic dataflow**
 - Also known as **functional dataflow**
 - That is what we will see now
 - It supports all the techniques of functional programming
- What are the two other paradigms?
 - **Message-passing concurrency** (e.g., **Erlang** and **Scala actors**)
 - Activities send messages to each other (like sending letters)
 - Relatively straightforward, can be combined with dataflow
 - **Shared-state concurrency** (e.g., **Java monitors**)
 - Activities share the same data and they try to work together without getting in each other's way
 - Much more complicated
 - Unfortunately, many current languages still use this paradigm

Later in the course

LINFO1131

© 2022 P. Van Roy. All rights reserved.

25

An unbound variable



- Let us explain dataflow by starting with an unbound variable
- An unbound variable is created in memory but not bound to a value
- What happens when you invoke an operation with an unbound variable?
local X Y **in**
 Y=X+1
 {Browse Y}
end
- What happens?

© 2022 P. Van Roy. All rights reserved.

26

What to do with an uninitialized variable?



- Different languages do different things
 - In **C**, the addition continues and X has a “garbage value” (= content of X’s memory at that moment)
 - In **Java**, the addition continues and X’s value is 0 (if X is an object attribute with type integer)
 - In **Prolog**, execution stops with an error
 - In **Java**, the compiler detects an error (if X is a local variable)
 - In **Oz**, execution waits just before the addition and continues when X is bound (dataflow execution)
 - In **constraint programming**, the equation “ $Y=X+1$ ” is added to the set of constraints and execution continues. An amazing way to compute!

LINFO2365

© 2022 P. Van Roy. All rights reserved.

27

Continuing the execution



- The waiting instruction:
declare X
local Y **in**
 $Y=X+1$
 {Browse Y}
end
- If someone would bind X, then execution could continue
- But who can do it?

© 2022 P. Van Roy. All rights reserved.

28

Continuing the execution

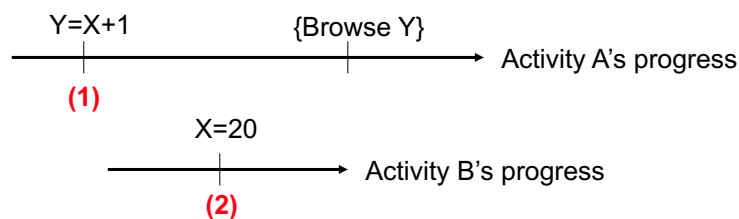


- The waiting instruction:
`declare X`
`local Y in`
 `Y=X+1`
 `{Browse Y}`
`end`
- If someone would bind X, then execution could continue
- But who can do it?
- Answer: another concurrent activity!
- If another activity does:
 `X=20`
- Then the addition will continue and display 21!
- This is called **dataflow execution**

© 2022 P. Van Roy. All rights reserved.

29

Dataflow execution



- Activity A waits patiently at point **(1)** just before the addition
- When activity B binds `X=20` at point **(2)**, then activity A can continue
- If activity B binds `X=20` **before** activity A reaches point **(1)**, then activity A **does not have to wait**

© 2022 P. Van Roy. All rights reserved.

30

Threads



- We add a language concept to support concurrent activities
 - In a program, an activity is a **sequence of executing instructions**
 - We add this concept to the language and call it a **thread**
- Each thread is **sequential**
- Each thread is **independent** of the others
 - There is no order defined between different threads
 - The system executes all threads using **interleaving semantics**: it is as if only one thread executes at a time, with execution stepping from one thread to another
 - The system guarantees that each thread receives a fair share of the computational capacity of the processor
- Two threads can communicate if they share a variable
 - For example, the variable corresponding to identifier X in the example we just saw

© 2022 P. Van Roy. All rights reserved.

31

Thread creation



- Creating a thread in Oz is simple
- Any instruction can be executed in a new thread:
thread <s> **end**
- For example:
declare X
thread {Browse X+1} **end**
thread X=1 **end**
- What does this small program do?
 - **Several executions are possible**, but they all eventually arrive at the same result: 2 is displayed!

© 2022 P. Van Roy. All rights reserved.

32



A small program (1)

- A small program with several threads:
declare X0 X1 X2 X3 **in**
thread X1=1+X0 **end**
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- The Browser displays [X0 X1 X2 X3]
 - The variables are all unbound
 - The Browser also uses dataflow:
when a variable is bound, the display is updated

© 2022 P. Van Roy. All rights reserved.

33



A small program (2)

- A small program with several threads:
declare X0 X1 X2 X3 **in**
thread X1=1+X0 **end**
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- Two threads will wait:
 - X1=1+X0 waits (since X0 is unbound)
 - X3=X1+X2 waits (since X1 and X2 are unbound)

© 2022 P. Van Roy. All rights reserved.

34



A small program (3)

- A small program with several threads:
declare X0 X1 X2 X3 **in**
thread X1=1+X0 **end**
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- Let's bind one variable
 - Bind X0=4

© 2022 P. Van Roy. All rights reserved.

35



A small program (4)

- A small program with several threads:
declare X0 X1 X2 X3 **in**
thread X1=1+X0 **end**
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- Let's bind one variable
 - Bind X0=4
 - The first thread executes and binds X1=5
 - The Browser displays [4 5 _ _]

© 2022 P. Van Roy. All rights reserved.

36



A small program (5)

- A small program with several threads:
declare X0 X1 X2 X3 **in**
thread X1=1+X0 **end** % terminated
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- The second thread is still waiting
 - Because X2 is still unbound

© 2022 P. Van Roy. All rights reserved.

37



A small program (6)

- A small program with several threads:
declare X0 X1 X2 X3 **in**
thread X1=1+X0 **end** % terminated
thread X3=X1+X2 **end**
{Browse [X0 X1 X2 X3]}
- Let's do another binding
 - Bind X2=7
 - The second thread executes and binds X3=12
 - The Browser displays [4 5 7 12]

© 2022 P. Van Roy. All rights reserved.

38

The Browser is a dataflow program



- The Browser executes with its own threads
- For each unbound variable that is displayed, there is a thread in the Browser that waits until the variable is bound
 - When the variable is bound, the display is updated
- This does not work with cells
 - The Browser uses the functional dataflow paradigm
 - The Browser does not look at the content of cells

© 2022 P. Van Roy. All rights reserved.

39

Streams and agents



© 2022 P. Van Roy. All rights reserved.

40

Streams



- A **stream** is a list that ends in an unbound variable
 - $S = a|b|c|d|S2$
 - A stream can be extended with new elements as long as necessary
 - The stream can be closed by binding the end to nil
- A stream can be used as a **communication channel** between two threads
 - The first thread adds elements to the stream
 - The second thread reads the stream

© 2022 P. Van Roy. All rights reserved.

41

Programming with streams



- This program displays the elements of a stream as they appear:

```
proc {Disp S}
  case S of X|S2 then {Browse X} {Disp S2} end
end
declare S
thread {Disp S} end
```
- We can add elements gradually:

```
declare S2 in S=a|b|c|S2
declare S3 in S2=d|e|f|S3
```
- Try it yourself!

© 2022 P. Van Roy. All rights reserved.

42

Producer/ consumer (1)

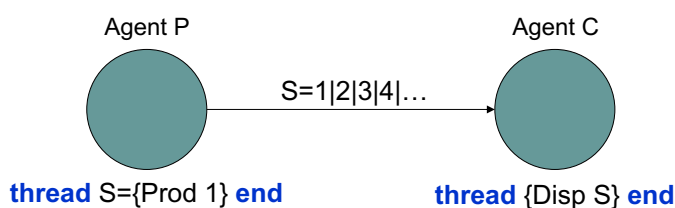


- A **producer** generates a stream of data
`fun {Prod N} {Delay 1000} N|{Prod N+1} end`
 - The {Delay 1000} slows down execution enough to observe it
- A **consumer** reads the stream and performs some action (like the Disp procedure)
- A producer/consumer program:
`declare S
thread S={Prod 1} end
thread {Disp S} end`

© 2022 P. Van Roy. All rights reserved.

43

Producer/ consumer (2)



- Each circle is a **concurrent activity that reads and writes streams**
 - We call this an **agent**
- Agents P and C communicate through stream S
 - The first thread creates the stream, the second reads it

© 2022 P. Van Roy. All rights reserved.

44

Pipeline (1)

- We can add more agents between P and C
- Here is a **transformer** that modifies the stream:

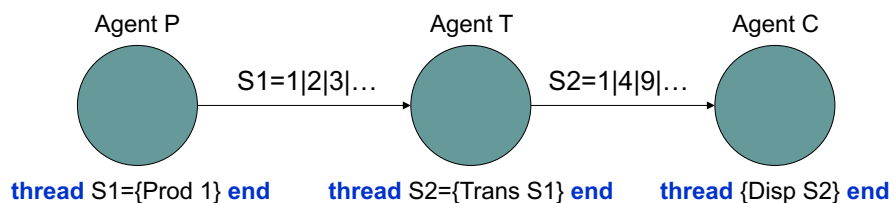
```
fun {Trans S}  
  case S of X|S2 then X*X|{Trans S2} end  
end
```
- This program has three agents:

```
declare S1 S2  
thread S1={Prod 1} end  
thread S2={Trans S1} end  
thread {Disp S2} end
```

© 2022 P. Van Roy. All rights reserved.

45

Pipeline (2)



- We now have three agents
 - The producer (agent P) creates stream S1
 - The transformer (agent T) reads S1 and creates S2
 - The consumer (agent C) reads S2
- The pipeline is a very useful technique!
 - For example, it is **omnipresent in operating systems since Unix**

© 2022 P. Van Roy. All rights reserved.

46

Agents



- An agent is a concurrent activity that reads and writes streams
 - The simplest agent is a list function executing in one thread
 - Since list functions are tail-recursive, the agent can execute with a fixed memory size
 - This is the deep reason why single assignment is important: it allows tail-recursive list functions, which makes deterministic dataflow into a practical paradigm
- All list functions can be used as agents
 - All functional programming techniques can be used in deterministic dataflow
 - Including higher-order programming! Later on we will see more examples of the power of the model.

© 2022 P. Van Roy. All rights reserved.

47

Thread semantics



© 2022 P. Van Roy. All rights reserved.

48

Thread semantics (1)



- We extend the abstract machine with threads
- Each thread has one semantic stack
 - The instruction **thread** <s> **end** creates a new stack
 - All stacks share the same memory
- There is **one sequence of execution states**, and threads take turns executing instructions
 - $(MST_1, \sigma_1) \rightarrow (MST_2, \sigma_2) \rightarrow (MST_3, \sigma_3) \rightarrow \dots$
 - MST is a multiset of semantic stacks
 - Each step " \rightarrow " executes one step in one thread
 - The choice of which thread to execute is made by the **scheduler**
 - This is called **interleaving semantics**

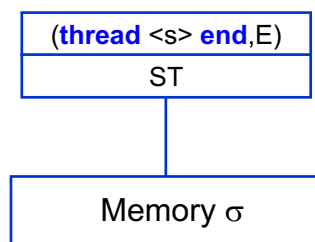
© 2022 P. Van Roy. All rights reserved.

49

Thread semantics (2)



A semantic stack that is about to create a thread



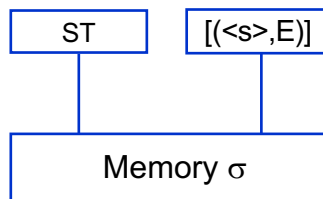
© 2022 P. Van Roy. All rights reserved.

50

Thread semantics (3)



We now have two stacks!



© 2022 P. Van Roy. All rights reserved.

51

Why interleaving semantics?



- **Interleaving semantics** is much easier to reason about than true concurrency semantics
 - **True concurrency semantics** = more than one thread can execute in one execution step
- Imagine that all threads execute in parallel, each with its own processor but all sharing the same memory
 - What happens when two threads write simultaneously at the same memory word?
 - With interleaving semantics, one thread will always write before the other, which makes reasoning simple
 - True concurrency semantics also models where threads "step on each others' toes", but usually this is not needed, since the hardware is careful to keep this from happening
 - For example, in a multicore processor the cache coherence protocol avoids simultaneous operations on one memory word

© 2022 P. Van Roy. All rights reserved.

52

Order of execution states



© 2022 P. Van Roy. All rights reserved.

53

Order of execution states



- In a sequential program, execution states are in a **total order**
 - **Total order** = when comparing any two execution states, one must happen before the other
- In a concurrent program, execution states of **one thread** are in a total order
 - The execution states of the complete program (with more than one thread) are in a **partial order**
 - **Partial order** = when comparing any two execution states, either one is before the other or there is no order between them
- In a concurrent program, **many executions** are compatible with the partial order
 - In the actual execution on the processor, **the scheduler chooses one execution (this choice is called nondeterminism)**

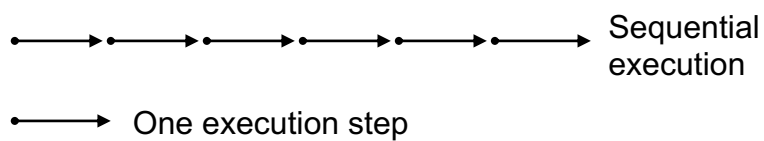
© 2022 P. Van Roy. All rights reserved.

54

Total order of a sequential program



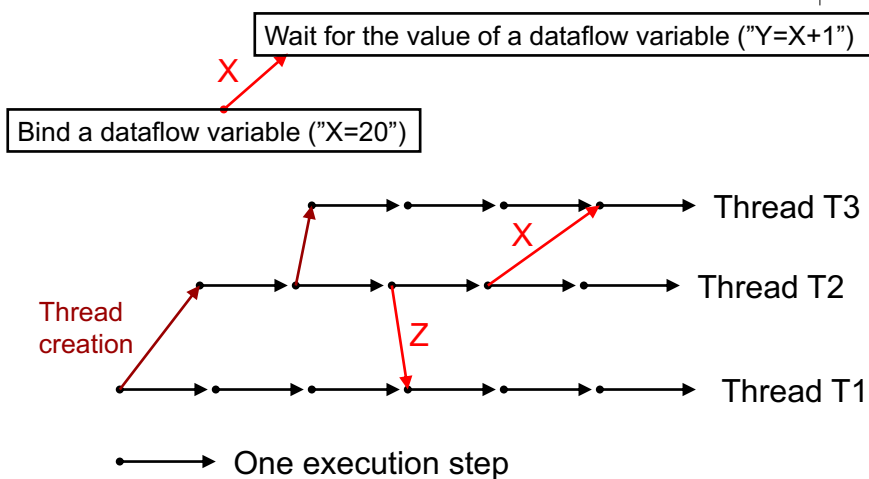
- In a sequential program, execution states are in a **total order**
- A sequential program has **one thread**



© 2022 P. Van Roy. All rights reserved.

55

Partial order of a concurrent program



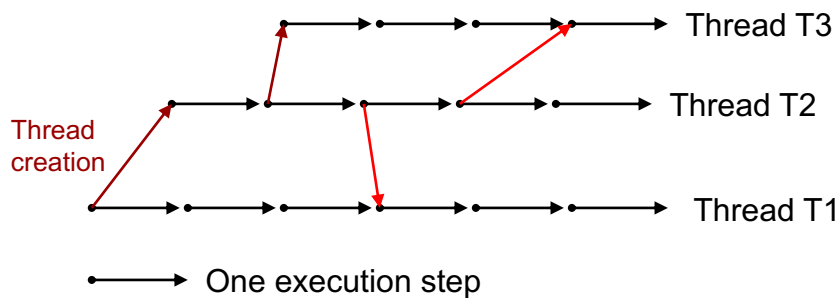
© 2022 P. Van Roy. All rights reserved.

56

Partial order of a concurrent program



- In a concurrent program, many executions are compatible with the partial order
- The scheduler chooses one of them during the actual execution (**nondeterminism**)



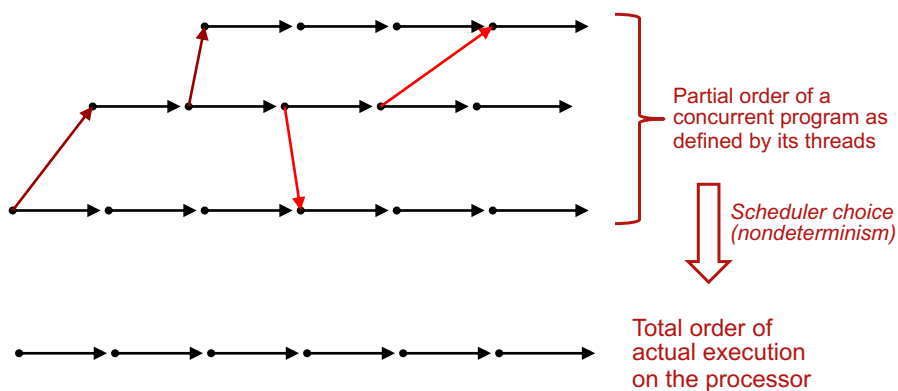
© 2022 P. Van Roy. All rights reserved.

57

The actual execution order



- The scheduler chooses the actual execution order, compatible with the partial order (**nondeterminism**)



© 2022 P. Van Roy. All rights reserved.

58

Nondeterminism



© 2022 P. Van Roy. All rights reserved.

59

Nondeterminism and the scheduler



- **Nondeterminism** is the ability of the system to make decisions independently of the application developer
 - The decisions can vary from one execution to the next
- The **scheduler** is the part of the system that decides at each moment which thread to execute
 - This decision is an example of **nondeterminism**
- Nondeterminism exists in all concurrent systems
 - It must be so, since the concurrent activities are **independent**
 - A crucial part of any concurrent program is how to manage its nondeterminism

© 2022 P. Van Roy. All rights reserved.

60

Example of nondeterminism (1)



- What does the following program do?

```
declare X
thread X=1 end
thread X=2 end
```
- The execution order of the two threads is not fixed
 - X will be bound to 1 or 2, we don't know which
 - The other thread **will have an error (raise an exception)**
 - A variable cannot be assigned to two values
- This is an example of **nondeterminism**
 - **A choice made by the system during execution**
 - The system is free to choose one or the other

© 2022 P. Van Roy. All rights reserved.

61

Example of nondeterminism (2)



- What does the following program do?

```
declare X={NewCell 0}
thread X:=1 end
thread X:=2 end
```
- The execution order of the two threads is not fixed
 - Cell X will first be bound to one value, then to the other
 - When both threads terminate, X will contain 1 or 2, we don't know which
 - This time there is no error
- This is an example of **nondeterminism**
 - **A choice made by the system during execution**

© 2022 P. Van Roy. All rights reserved.

62

Example of nondeterminism (3)



- What does the following program do?

```
declare X={NewCell 0}  
thread X:=1 end  
thread X:=1 end
```
- It makes a choice, just like the previous program
 - But in this case, the final results are the same (by accident)
- **This is still nondeterminism!**
 - The important point is the **choice**: the running program still sees a difference in the threads' execution order
 - Maybe the results are the same by accident (depending on the computations done), but the choice remains

© 2022 P. Van Roy. All rights reserved.

63

Managing nondeterminism



- Nondeterminism *must always be managed*
 - It should not affect program correctness
 - The most complicated case is when **threads and cells are used in the same program** (see previous example)
 - Unfortunately, this is exactly how many languages handle concurrency
- Deterministic dataflow has a major advantage
 - **The result of a program is always the same** (except if there is a programming error – if a thread raises an exception)
 - The nondeterminism of the scheduler **does not affect the result**
 - There is no **observable** nondeterminism

© 2022 P. Van Roy. All rights reserved.

64



How the scheduler works (1)

- If the number of threads is larger than the number of processors (usually true), then threads will share the processors
 - Each thread is executing during a short time period that is called a **time slice**
- The choice of which thread to execute and for how long is made by the scheduler
- A thread is **runnable** if the instruction on the top of its stack is not waiting on a dataflow variable. Otherwise, the thread is **suspended**, in other words **blocked on a variable**.

© 2022 P. Van Roy. All rights reserved.

65



How the scheduler works (2)

- A scheduler is **fair** if every runnable thread will eventually (= in finite time) be executed
 - Usually, threads are classified according to their **priority**, and some **additional guarantees** are given on the percentage of the processor time that is given to the threads of the same priority
- If the scheduler is fair, then it is possible to reason about program execution (all programs will run)
- If the scheduler is not fair, a perfectly correct program may not run correctly
 - Certain threads may **starve**, i.e., receive 0% of the processor time, so they never execute

© 2022 P. Van Roy. All rights reserved.

66

“Concurrency for dummies”



© 2022 P. Van Roy. All rights reserved.

67

“Concurrency for dummies”



- The multi-agent programs we saw so far are **deterministic**
 - Their nondeterminism is not observable (results always the same)
 - The agent Trans with input 1|2|3|_ always outputs 1|4|9|_
- In these programs, concurrency does not change the result but only **the order in which computations are done** (that is, **when** the result is calculated)
 - It is possible to add threads at will to a program without changing the result (we call this **Concurrency for Dummies**)
 - The only effect of added threads is to make the program more incremental (by interleaving execution and removing deadlocks)
- Only possible in **functional programming** (deterministic dataflow!)
 - It is not true when using cells and threads together (Java!)
 - It is a consequence of the Church-Rosser theorem

© 2022 P. Van Roy. All rights reserved.

68



Example (1)

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

© 2022 P. Van Roy. All rights reserved.

69



Example (2)

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

© 2022 P. Van Roy. All rights reserved.

70



Example (3)

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

thread ... end
can be used as
an expression

© 2022 P. Van Roy. All rights reserved.

71



Example (4)

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

- What happens when we execute:
declare F
{Browse {CMap [1 2 3 4] F}}

© 2022 P. Van Roy. All rights reserved.

72



Example (5)

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

```
declare F
{Browse {CMap [1 2 3 4] F}}
```

- The Browser displays [_ _ _ _]
 - CMap calculates a list with unbound variables
 - The new threads wait until F is bound
- What would happen if {F X} was not in its own thread?
 - Nothing would be displayed! The CMap call would block.

© 2022 P. Van Roy. All rights reserved.

73



Example (6)

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

- What happens when we bind F:
F = fun {\$ X} X+1 end

© 2022 P. Van Roy. All rights reserved.

74



Example (7)

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

- The Browser displays [2 3 4 5]
- With or without the thread creation, the final result is always [2 3 4 5]

© 2022 P. Van Roy. All rights reserved.

75



Concurrency for dummies!

- Threads can be added at will to a functional program **without changing the result**
- Therefore it is very easy to take a functional program and make it concurrent
- It suffices to insert **thread ... end** in those places that need concurrency
- **Warning:** concurrency for dummies does not work in a program with explicit state (= with cells!)
 - For example, it does not work in Java
 - In Java, concurrency is handled with the concept of a **monitor** (= **synchronized object**), which coordinates how multiple threads access an object. This is *much more complicated* than deterministic dataflow.

© 2022 P. Van Roy. All rights reserved.

76

Why does it work? (1)



```
fun {Fib X}
  if X==0 then 0
  elseif X==1 then 1
  else
    thread {Fib X-1} end + {Fib X-2}
  end
end
```

© 2022 P. Van Roy. All rights reserved.

77

Why does it work? (2)



```
fun {Fib X}
  if X==0 then 0 elseif X==1 then 1
  else F1 F2 in
    F1 = thread {Fib X-1} end
    F2 = {Fib X-2}
    F1 + F2
  end
end
```

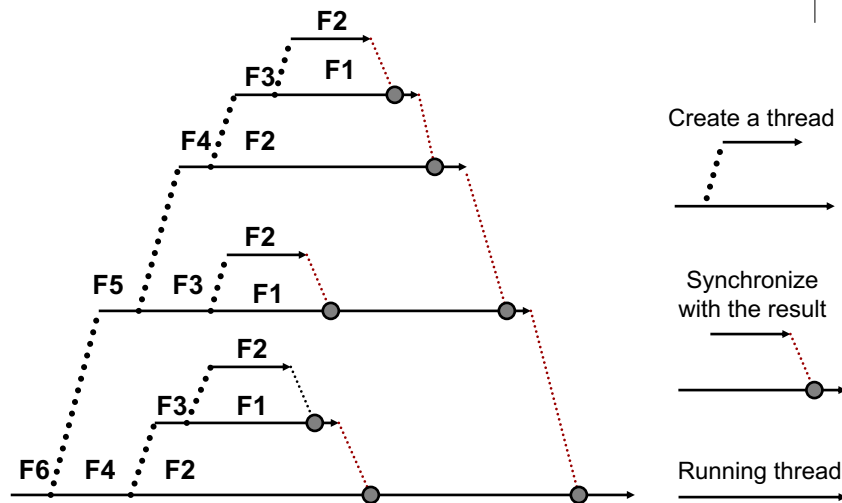
Dataflow dependency

It works because variables can only be bound to one value (single assignment)

© 2022 P. Van Roy. All rights reserved.

78

Execution of {Fib 6}

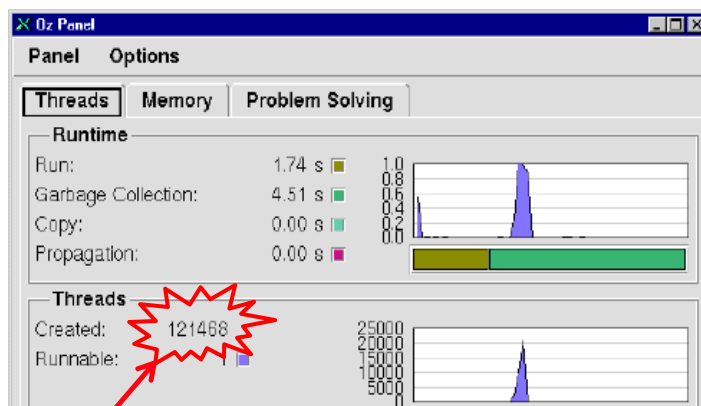


© 2022 P. Van Roy. All rights reserved.

79

Observing the execution of Fib

Only in Mozart 1



Total number of threads created since system startup

© 2022 P. Van Roy. All rights reserved.

80

Counting threads

C={NewCell 0}

proc {Inc C}

{Exchange C X Y} Y=X+1

end

fun {Fib X}

if X==0 **then** 0

elseif X==1 **then** 1

else

thread {Inc C} {Fib X-1} **end** + {Fib X-2}

end

end

This works also in Mozart 2

C:= @C + 1 is not correct!

It is because the scheduler can put the thread to sleep after the X=@C+1 and before the C:=X.

© 2022 P. Van Roy. All rights reserved.

81

Multi-agent programming

© 2022 P. Van Roy. All rights reserved.

82

Multi-agent programming

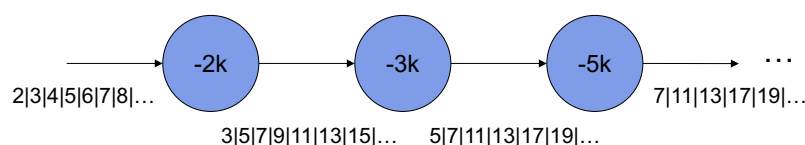


- Earlier in the course we saw some simple examples of multi-agent programs
 - Producer/consumer
 - Producer/transformer/consumer (pipeline)
- Let's see two more sophisticated examples
 - **Sieve of Eratosthenes**: dynamically building a pipeline during its execution
 - **Digital logic simulation**: using higher-order programming together with concurrency

© 2022 P. Van Roy. All rights reserved.

83

The Sieve of Eratosthenes



- The Sieve of Eratosthenes is an algorithm for calculating a sequence of prime numbers
- Each agent in the pipeline removes multiples of an integer
- Starting with a sequence containing all integers, we end up with a sequence of primes

© 2022 P. Van Roy. All rights reserved.

84



A filter agent

- A list function that removes multiples of K:

```
fun {Filter Xs K}
  case Xs of X|Xr then
    if X mod K \= 0 then X|{Filter Xr K}
    else {Filter Xr K} end
  else nil
  end
end
```

- We make an agent by putting it in a thread:

```
thread Ys={Filter Xs K} end
```

© 2022 P. Van Roy. All rights reserved.

85



The Sieve program

- Sieve builds the pipeline during execution:

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

declare Xs Ys in
  thread Xs={Prod 2} end
  thread Ys={Sieve Xs} end
{Browse Ys}
```

Concurrent deployment

Building the infrastructure of a concurrent program during its execution (execution will just wait if a part that it needs is not built yet)

© 2022 P. Van Roy. All rights reserved.

86

An optimization



- Otherwise too many do-nothing agents are created!

```
fun {Sieve2 Xs M}  
  case Xs  
  of nil then nil  
  [] X|Xr then  
    if X=<M then  
      X|{Sieve2 thread {Filter Xr X} end M}  
    else Xs end  
  end  
end
```

- We call {Sieve2 Xs 316} to generate a list of primes up to 100000 (why?)

© 2022 P. Van Roy. All rights reserved.

87

Digital logic simulation



© 2022 P. Van Roy. All rights reserved.

88

Digital logic simulation



- The deterministic dataflow paradigm makes it easy to model digital logic circuits
- We show how to model combinational logic circuits (no memory) and sequential logic circuits (with memory)
- Signals in time are represented as streams; logic gates are represented as agents

© 2022 P. Van Roy. All rights reserved.

89

Modeling digital circuits



- Real digital circuits consist of active circuit elements called gates which are interconnected using wires that carry digital signals
- A **digital signal** is a voltage in function of time
 - Digital signals are meant to carry two possible values, called 0 and 1, but they may have noise, glitches, ringing, and other undesirable effects
- A **digital gate** has input and output signals
 - The output signal is slightly delayed with respect to the input
- We will model **gates as agents** and **signals as streams**
 - This assumes perfectly clean signals and zero gate delay
 - We will later add a delay gate in order to model gate delay

© 2022 P. Van Roy. All rights reserved.

90

Digital signals as streams



- A signal is modeled by a stream that contains elements with values 0 or 1

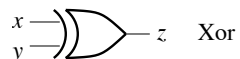
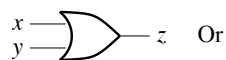
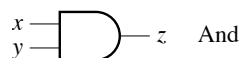
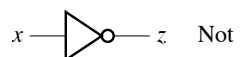
$$S = a_0 | a_1 | a_2 | \dots | a_i | \dots$$

- Time instants are numbered from when the circuit starts running
- At instant i , the signal's value $a_i \in \{0, 1\}$

© 2022 P. Van Roy. All rights reserved.

91

Digital logic gates



x	y	z			
		Not	And	Or	Xor
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

- Some typical logic gates with their standard pictorial symbols and the boolean functions that define them
- But gates are not just boolean functions!

© 2022 P. Van Roy. All rights reserved.

92

Digital gates as agents



- A gate is much more than a boolean function; it is an active entity that takes input streams and calculates an output stream

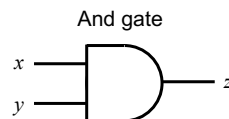
```

fun {And A B} if A==1 andthen B==1 then 1 else 0 end end
fun {Loop S1 S2}
  case S1#S2 of (A|T1)#(B|T2) then {And A B}|{Loop T1 T2} end
end
thread Sc={Loop Sa Sb} end
  
```

- Example execution:

```

Sx=0|1|0|Tx % input signal x
Sy=1|1|0|Ty % input signal y
Sz=0|1|0|Tz % output signal z
  
```



© 2022 P. Van Roy. All rights reserved.

93

Creating many gates



- Let us define an **abstraction** for building all the different kinds of logic gates we need
 - We define the function GateMaker that takes a two-argument boolean function Fun, where {GateMaker Fun} returns a function FunG that creates gates
 - Each call to FunG creates a running gate based on Fun
- This gives **three levels of abstraction** that we can compare with object-oriented programming:
 - GateMaker is analogous to a **generic class** or **metaclass**
 - FunG is analogous to a **class**
 - A running gate is analogous to an **object**

© 2022 P. Van Roy. All rights reserved.

94

GateMaker implementation



- Calling {GateMaker F} creates a gate maker:

```
fun {GateMaker F}  
  fun {$ Xs Ys}  
    fun {GateLoop Xs Ys}  
      case Xs#Ys of (X|Xr)#(Y|Yr) then  
        {F X Y}{{GateLoop Xr Yr}  
      end  
    end  
  in  
    thread {GateLoop Xs Ys} end  
  end  
end
```

© 2022 P. Van Roy. All rights reserved.

95

Making gates



- Each of these functions can make gates:

```
AndG={GateMaker fun {$ X Y} X*Y end}  
OrG={GateMaker fun {$ X Y} X+Y-X*Y end}  
NandG={GateMaker fun {$ X Y} 1-X*Y end}  
NorG={GateMaker fun {$ X Y} 1-X-Y+X*Y end}  
XorG={GateMaker fun {$ X Y} X+Y-2*X*Y end}
```

© 2022 P. Van Roy. All rights reserved.

96

Combinational logic



© 2022 P. Van Roy. All rights reserved.

97

Combinational logic



- Combinational logic has no memory: all calculation is done at the same time instant
- A gate is a simple combinational function:

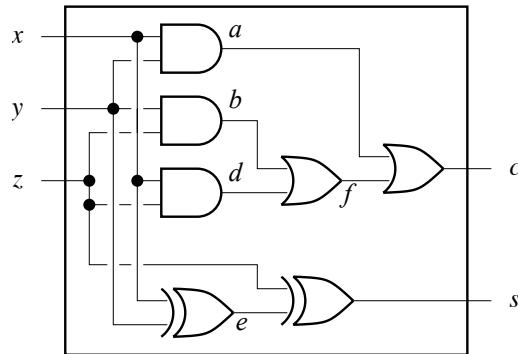


- Therefore, any number of interconnected gates also defines a combinational function
- We define a useful circuit called a **full adder**

© 2022 P. Van Roy. All rights reserved.

98

Full adder specification



x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- A full adder adds three 1-bit binary numbers x , y , and z giving a sum bit s and carry bit c
- An n -bit adder can be built by connecting n full adders

© 2022 P. Van Roy. All rights reserved.

99

Full adder implementation



- Full adder creation as five-argument component:

```

proc {FullAdder X Y Z C S}
  A B D E F
in
  A={AndG X Y}
  B={AndG Y Z}
  D={AndG X Z}
  F={OrG B D}
  C={OrG A F}
  E={XorG X Y}
  S={XorG Z E}
end
  
```

© 2022 P. Van Roy. All rights reserved.

100

Sequential logic



© 2022 P. Van Roy. All rights reserved.

101

Sequential logic

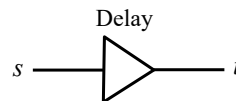


- Sequential logic has memory: past values of a signal influence the present values
- We add a way for the past to influence the present: a Delay gate

$S = a_0|a_1|a_2|\dots|a_i|\dots$

$T = b_0|b_1|b_2|\dots|b_i|\dots$

$b_i = a_{i-1} \Rightarrow T = 0|S$

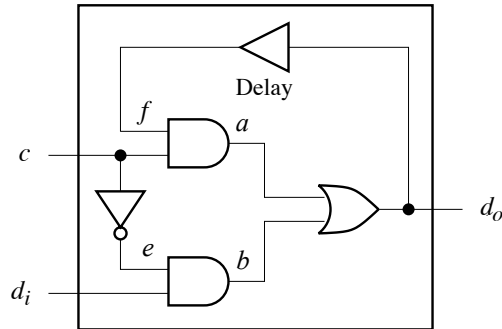


fun {DelayG S} 0|S **end**

© 2022 P. Van Roy. All rights reserved.

102

Latch specification



- A latch is a simple circuit with memory; it has two stable states and can memorize its input
- Output d_o follows input d_i and freezes when c is 1

© 2022 P. Van Roy. All rights reserved.

103

Latch implementation

- Latch creation as a three-argument component:

```

proc {Latch C Di Do}
  A B E F
in
  F={DelayG Do}
  A={AndG C F}
  E={NotG C}
  B={AndG E Di}
  Do={OrG A B}
end
  
```

© 2022 P. Van Roy. All rights reserved.

104

Summary and history



© 2022 P. Van Roy. All rights reserved.

105

Deterministic dataflow summary



- We have introduced a simple and expressive paradigm for concurrent programming
 - We can build multi-agent programs using **streams** (list with unbound tail) and **agents** (list function running in a thread)
- It is based on two simple ideas
 - **Single-assignment variables** that synchronize on binding
 - **Threads** that define a sequence of executing instructions
- By design, it has **no observable nondeterminism** (no race conditions)
 - Deterministic dataflow is a form of functional programming
 - « Concurrency for Dummies »

© 2022 P. Van Roy. All rights reserved.

106

Historical note: concurrency *must* get simpler

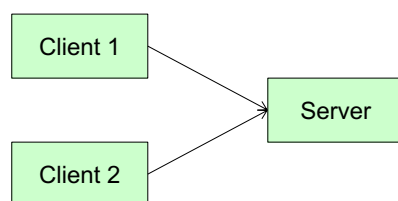


- Parallel programming has finally arrived (a surprise to old timers like me!)
 - **Multicore processors**: dual and quad today, a dozen tomorrow, a hundred in a decade, soon most apps will do it
 - **Distributed computing**: data-intensive with tens of nodes today (NoSQL, MapReduce), hundreds and thousands tomorrow, most apps will do it
- Something fundamental will have to change
 - Sequential programming can't be the default (it's a centralized bottleneck)
 - Libraries can only hide so much (interface complexity, distribution structure)
- Concurrency **must become easy**
 - Deterministic dataflow is functional programming!
 - It can be extended cleanly to distributed computing
 - Open network transparency
 - Modular fault tolerance
 - Large-scale distribution

© 2022 P. Van Roy. All rights reserved.

107

But is determinism the right default? Yes!



A client/server can't be written in a deterministic paradigm!

It's because the server must accept requests nondeterministically from the two clients

- Deterministic dataflow has strong limitations!
 - A program that needs nondeterminism can't be written
 - Even a simple **client/server can't be written**
- But determinism has enormous advantages, so it is the correct default
 - **Race conditions are impossible** by design
 - With determinism as default, we can **reduce the need for nondeterminism** (in the client/server, it's needed only at the point where the server accepts requests)
 - **Any functional program can be made concurrent** without changing the result

Not a problem!
Just add nondeterminism exactly where it is needed

© 2022 P. Van Roy. All rights reserved.

108

History of deterministic dataflow



- **Deterministic concurrency** has a long history that starts in 1974
 - Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress*, pp. 471-475, 1974. *Deterministic concurrency*.
 - Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pp. 993-998, 1977. *Lazy deterministic concurrency*.
- Why was it forgotten for so long?
 - Message passing and monitors arrived at about the same time:
 - Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 235-245, Aug. 1973.
 - Charles Antony Richard Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549-557, Oct. 1974.
 - **Actors and monitors express nondeterminism, so they are better. Right?**
- **Dataflow computing** also has a long history that starts in 1974
 - Jack B. Dennis. First version of a data flow procedure language. *Springer Lecture Notes in Computer Science*, vol. 19, pp. 362-376, 1974.
 - **Dataflow remained a fringe subject since it was always focused on parallel programming,** which only became mainstream with the arrival of multicore processors in mainstream computing (e.g., IBM POWER4, the first dual-core processor, in 2001).

© 2022 P. Van Roy. All rights reserved.