# LINFO1104
# Concepts, Paradigms, and Semantics of Programming Languages

## Course slides

Spring 2020

Peter Van Roy
Université catholique de Louvain
B-1348 Louvain-la-Neuve
Belgium

# LINFO1104
# Concepts, paradigms, and semantics of programming languages

## Lecture 1

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

## Context of this course

- LEPL1402: First semester course for all engineering students
  - Introduction to programming based on Java and IntelliJ
  - Java and object-oriented programming
  - Algorithms, data structures, and invariants
  - Complexity and software engineering
  - Introduction to functions and concurrency in Java
- LINFO1104: Second semester course for informatics majors
  - Programming paradigms based on Oz multiparadigm language
  - Formal semantics including lambda calculus
  - Procedural abstraction (higher-order programming)
  - Data abstraction (objects and abstract data types)
  - Symbolic programming
  - Concurrent programming
  - Multi-agent programming and Erlang

2

# Programming paradigms

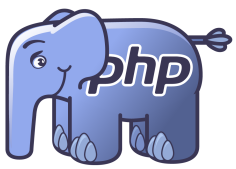
3

## *Hundreds* of programming languages are in use...







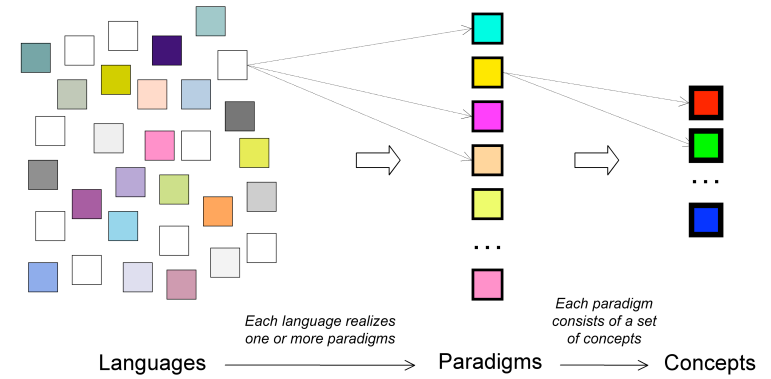






4

# So many, how can we understand them all?




- Key insight: languages are based on paradigms, and there are many fewer paradigms than languages
- We can understand *many* languages by learning *few* paradigms!

5

# What is a paradigm?

- A programming paradigm is an approach to programming a computer based on a coherent set of principles or a mathematical theory
- A program is written to solve problems
  - Any realistic program needs to solve different kinds of problems
  - Each kind of problem needs its own paradigm
  - So we need multiple paradigms and we need to combine them in the same program

6

# How can we study *multiple* paradigms?

- How can we study multiple paradigms without studying multiple languages (since most languages only support one, or sometimes two paradigms)?
- Each language has its own syntax, its own semantics, its own system, and its own quirks
  - Picking many languages, like Java, Erlang, Scheme, and Haskell, and structuring our course around them would be complicated
- Our pragmatic solution: we use two languages, Oz (a multiparadigm research language) and Erlang (a multi-agent industrial language)
  - Our textbook, *Concepts, Techniques, and Models of Computer Programming*, uses Oz to cover many paradigms



7

# How can we *combine* paradigms in a program?

- Each paradigm is a different way of thinking
  - How can we combine different ways of thinking in one program?
- We can do it using the concept of a kernel language
  - Each paradigm has a simple core language, its kernel language, that contains its essential concepts
    - Every practical language, even complicated ones, can be translated easily into its kernel language
  - Even very different paradigms have kernel languages that have much in common; often there is only one concept difference
- We start with a simple kernel language that underlies our first paradigm, functional programming
  - We then add concepts one by one to give the other paradigms
  - Scientific method: understand a system in terms of its parts

8

# Summary of the approach

- Hundreds of languages are used in practice: we cannot study them all in one course or in one lifetime
  - Solution: **focus on paradigms**, since each language is based on a paradigm and there there are many fewer paradigms than languages
- One language per paradigm is too much to study in a course, since each language is already complicated by itself
  - Solution: **use one research language**, Oz, that can express many paradigms
- Realistic programs need to combine paradigms, but how can we do it since each paradigm is a different way of thinking?
  - Solution: **define paradigms using kernel languages**, since different paradigms have kernel languages that are almost the same
  - Kernel languages allow us to define many paradigms by focusing on their differences, which is much more economical in time and effort

9

# Let's get started

- You should already know an object-oriented language!
  - Object-oriented programming, as used in Java, is clearly an important paradigm (as seen in LEPL1402)
  - But what about the other paradigms?
- Isn't object-oriented programming by far the most important and useful paradigm?
  - Actually, no, it's not!
  - Many other paradigms are extremely useful, often more so than OOP! For example, to make robust and efficient distributed programs on the Internet, OOP just does not solve the right problems. Multi-agent programming is much better for that, which is why we will give an introduction to Erlang.
  - LINFO1104 covers five paradigms that solve many problems

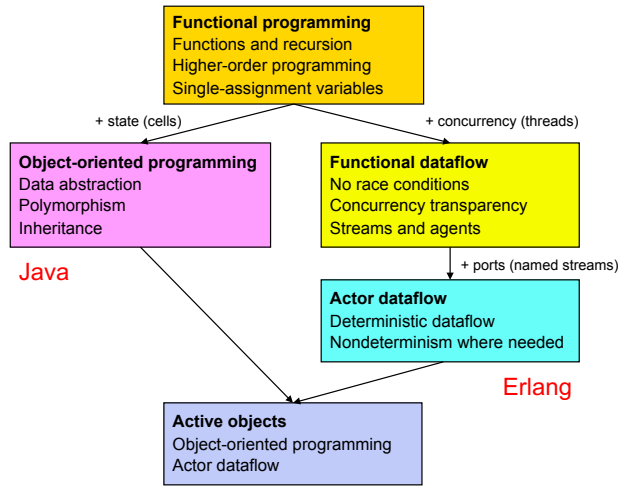
10

# *Five* paradigms

- LINFO1104 covers five main paradigms:
  - Functional programming
  - Object-oriented programming (Java)
  - Functional dataflow programming
  - Actor dataflow programming (Erlang)
  - Active objects
- These are probably the most important programming paradigms for general use
  - But there are many other paradigms, made for other problems: LINFO1104 gives you a good foundation for studying them later if you wish

11

# Five paradigms




12

# Our first paradigm

- Functional programming
  - It is the simplest paradigm
  - It is the foundation of all other paradigms
  - It is a form of *declarative programming: say what, not how*
- Our approach to functional programming
  - It is our first introduction to programming concepts
  - It is our first introduction to a kernel language
  - We use it to explain invariant programming
  - We use it to explain symbolic programming
  - We use it to explain higher-order programming
  - We give a formal semantics based on the kernel language

13

# Practical organization

14

# Course organization

- Weekly lecture
  - Tuesday 14h00-16h00 SUD08
- Weekly lab session
  - Tuesday 16h15-18h15 PCUR04 (except Week 1)
  - Wednesday 8h30-10h30 PCUR05
  - Wednesday 10h45-12h45 PCUR02
  - Wednesday 14h00-16h00 PCUR03
  - Monday 14h00-16h00 PCUR01
- Midterm exam
  - Optional, counts for 5 points on final grade (max of midterm & final Q1)
- Course project
  - Practical programming project in groups of two students
  - Mandatory, counts for 5 points on final grade
  - During last third of semester

15

# Software support

- LINFO1104 Moodle
  - All practical information is given here
- Mozart Programming System
  - For practical exercises; www.mozart2.org
  - Download and install Mozart 2.0.1
- Erlang OTP
  - For multi-agent programming; www.erlang.org
  - Download and install OTP 22.2 (in second half of course)

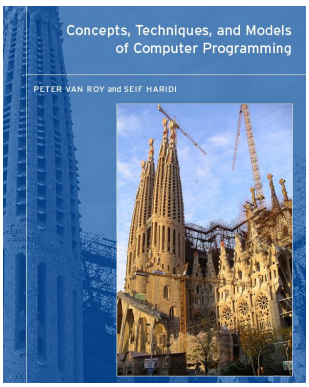
16

# Educational team

- Professor
  - Peter Van Roy peter.vanroy@uclouvain.be
- Teaching assistants
  - Gorby Kabasele gorby.kabasele@uclouvain.be
  - Antoine Vanderschueren antoine.vanderschueren@uclouvain.be
  - Thomas Wirtgen thomas.wirtgen@uclouvain.be
- Tutors
  - Florence Blondiaux
  - Sébastien Kalbusch
  - Hadrien Libioulle
  - Lucas Maris
  - Mohammad Zareie

17

# Course textbook and slides




- "Concepts, Techniques, and Models of Computer Programming" by Peter Van Roy and Seif Haridi, MIT Press
- Slides for each week's lecture
  - The slides contain most of the course material, but please look at the book if anything is unclear or if you want more examples

18

# Basic concepts

19

# Interactive system

**declare**

X = 1234 * 5678

{Browse X}

- Select a region in the Emacs buffer
- Feed the region to the system
  - The text is compiled and executed
- Interactive system can be used as a powerful calculator →

20

# Creating variables

**declare**

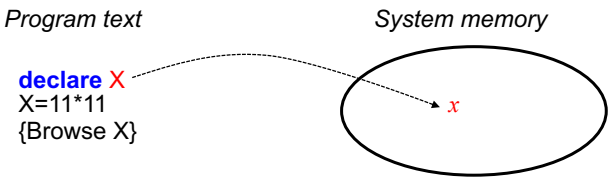
X = 1234 * 5678

{Browse X}

- **Declare** (create) a variable **designated** by X
- **Assign** to the variable the value 7006652
  - Result of the calculation 1234*5678
- **Call** the procedure Browse with the argument designated by X
  - Opens a window that displays 7006652

21

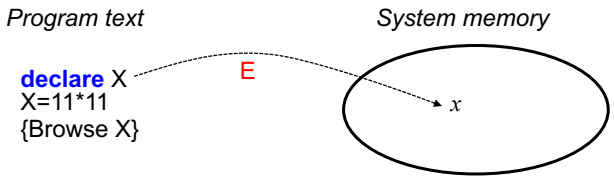
# Identifiers and variables




- There are two concepts hiding in plain view here
  - Identifier X : what you type (character sequence starting with uppercase) Var, A, X123, FirstCapitalBank
  - Variable $x$ : what is in memory (used to store the value)
- Variables are short-cuts for values (= constants)
  - Can only be assigned to one value (like mathematical variables)
  - Multiple assignment is another concept! We will see it later in the course.
  - The type of the variable is only known when it is assigned (dynamic typing)

22

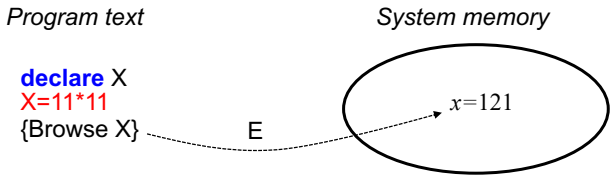
# Environment




- **declare** is an interactive instruction
  - Creates a new variable in memory
  - Links the identifier and its corresponding variable
- Third concept: environment E={X→ $x$}
  - A function that takes an identifier and returns a variable: E(X) = $x$
  - Links identifiers and their corresponding variables (and the values they are bound to)

23

# Assignment




- The assignment instruction X=121 binds the variable $x$ to the value 121

24

# Single assignment

- A variable can only be bound to one value
  - It is called a single-assignment variable
  - Why? Because we are doing functional programming!
- Incompatible assignment: signals an error
  X = 122
- Compatible assignment: accepted
  X = 121

25

# Why single assignment?

- Single assignment is part of functional programming
  - It means that variables are *mathematical variables*, like in an equation
- Programming with mathematical variables is easy
  - It seems like a big handicap, not being able to assign twice
  - It is actually quite easy, as we will see
- Advantages
  - Programs are much easier to prove, analyze, test, and debug
  - All programming languages are based on functional programming
- Disadvantages
  - The main disadvantage is that interaction with the real world is harder (e.g., with human users that connect to the system and interact)
  - We will solve this problem later by adding one new concept, namely multiple assignment (like Java variables)

26

# The functional style can be done in all languages

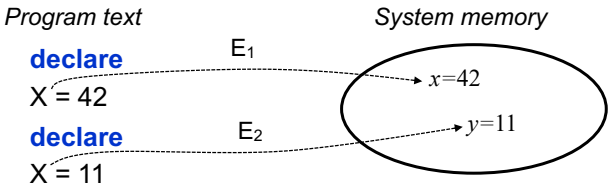
- "A program that works today will work tomorrow"
  - Functions and variables don't change
  - All changes are in the arguments, not in the functions
- This programming style is encouraged in all languages (incl. Java)
  - "Stateless server" for a client/server application
  - "Stateless component" for a service application

"Cookies" on the Web

- Learning functional programming helps us think in this style
  - As well as helping us understand all programming languages, today's and tomorrow's

27

# Redeclaring an identifier




- An identifier can be redeclared
  - The same identifier refers to a different value
  - There is no conflict with single assignment.
    Each occurrence of X corresponds to a different variable.
- The interactive environment always has the last declaration
  - **declare** keeps the same correspondance until redeclared (if ever)
  - In this example X will refer to 11

28

# Scope of an identifier occurrence

```
local
   X
in
   X = 42  {Browse X}
   local
      X
   in
      X = 11  {Browse X}
   end
   {Browse X}
end
```

- The instruction
  **local** X **in** <stmt> **end**
  declares X between **in** and **end**
- The scope of an identifier occurrence is that part of the program text for which the occurrence corresponds to the same variable declaration
- The scope can be determined by inspecting the program text; no execution is needed. This is called lexical scoping or static scoping.
- Why is there no conflict between X=42 and X=11, even though variables are single assignment?
- What will the third Browse display?

29

# Tips on Oz syntax

30

# Tips on Oz syntax

- You can see that Oz syntax is not like Java syntax
- The most popular syntax in mainstream languages (Java and C++) is « C-like », where identifiers are statically typed (« int i; ») and start with lowercase, and code blocks are delimited by braces { … }
  - Oz syntax is definitely not C-like!
- Oz syntax is designed for multiparadigm programming
  - Oz syntax is inspired by many languages: Prolog (logic programming), Scheme and ML (functional programming), C++ and Smalltalk (object-oriented programming), and so forth
- (Later on we will see Erlang, with even another syntax)

31

# Why is Oz syntax different?

- It is different because Oz supports many programming paradigms
  - The syntax is carefully designed so that the paradigms don't interfere with each other
  - It's possible to program in just one paradigm. It's also possible to program in several paradigms that are cleanly separated in the program text.
- So it is important not to get confused by the differences between Oz syntax and other syntaxes you may know
- We show the main differences so that you will not be hindered by them

32

# Four main differences in Oz syntax

- Identifiers in Oz always start with an uppercase letter
  - Examples: X, Y, Z, Min, Max, Sum, IntToFloat.
  - Why? Because lowercase is used for symbolic constants (atoms).
- Procedure and function calls in Oz are surrounded by braces { ... }
  - Examples: {Max 1 2}, {SumDigits 999}, {Fold L F U}.
  - Why? Because parentheses are used for record data structures.
- Local identifiers are introduced by **local** ... **end**
  - Example: **local** X **in** X=10+20 {Browse X} **end**.
  - Why? Because all compound instructions in Oz start with a keyword (such as « **local** ») and terminate with **end**.
- Variables in Oz are single assignment
  - Examples: **local** X Y **in** X=10 Y=X+20 {Browse Y} **end**.
  - Why? Because the first paradigm is functional programming. Multiple assignment is a concept that we will introduce later.

33

# Oz syntax in the programming exercises

- Most programming bugs, at least early on, are due to syntax errors
  - Most common error: lowercase letter to start an identifier
- Please take into account the four main differences. Once you have assimilated them, reading and writing Oz will become straightforward.
- And now let's introduce functions

34

# Functional programming

35

## Functions

- We would like to execute the same code many times, each time with different values for some of the identifiers
  - To avoid repeating the same program code, we can create a function
- A function defines program code to execute
  - A function is just another kind of value in memory, like a number (as we will see later)
  - Variables can be bound to functions just as easily as to numbers
- The function Sqr returns the square of its input:

```
declare
fun {Sqr X} X*X end
```

- The **fun** keyword identifies the function. The identifier Sqr refers to a variable that is bound to the function.

36

# Numbers

- There are two kinds of numbers in Oz
  - Exact numbers: integers
  - Approximate numbers: floating point
- Integers are exact (arbitrary precision)
- Floats are approximations of real numbers (around 15 digits precision, using 64-bit representation)
- There is never any automatic conversion from exact to approximate and vice versa
  - To convert, we use functions IntToFloat or FloatToInt
  - Design principle: don't mix incompatible concepts

37

# SumDigits3

- Function SumDigits3 calculates the sum of digits of a three-digit positive integer:

```
declare
fun {SumDigits3 N}
   (N mod 10) + ((N div 10) mod 10) +
   ((N div 100) mod 10)
end
```

- **mod** and **div** are integer functions
- / (division) is a float function
- * (multiplication) is a function on both floats and integers

38

## SumDigits6

- Sum of digits of a six-digit positive integer

```
fun {SumDigits6 N}
   {SumDigits3 (N div 1000)} +
   {SumDigits3 (N mod 1000)}
end
```

- This is an example of function composition: defining a function in terms of other functions
  - This is a key ability for building large systems: we can build them in layers, where each layer is built by a different person
  - This is the first step towards data abstraction

39

## SumDigitsR (first try)

- Sum of digits of any positive integer
- We use recursion: the function calls itself with a smaller argument

```
fun {SumDigitsR N}
   (N mod 10) + {SumDigitsR (N div 10)}
end
```

- This function calls itself with a smaller value
  - But it never stops: we need to make it stop!

40

# SumDigitsR (correct)

- Sum of digits of any positive integer

```
fun {SumDigitsR N}
   if (N==0) then 0
   else
      (N mod 10) + {SumDigitsR (N div 10)}
   end
end
```

- This introduces the conditional (**if**) statement
- This is a correct example of function recursion: defining a function that calls itself
  - This is a key ability for building complex algorithms: we divide a complex problem into simpler subproblems (divide and conquer)

41

# We can now do functional programming

- Our small language can do functional programming
  - Calculating with numbers, functions, and conditionals can do everything!
- This language is Turing complete
  - We can compute anything that any computer can compute
- Functional programming supports powerful techniques
  - Invariant programming, symbolic programming
  - Higher-order programming, data abstraction (which includes objects)
  - Concurrent programming
- Is there an even smaller language for functional programming?
  - Yes, the lambda calculus is the smallest!
  - We will see the lambda calculus later in the course
  - All programming languages are based on the lambda calculus

42

# Invariant programming

43

## Invariant programming is for loops

- A loop is a part of a program that is repeated until a condition is satisfied
  - Loops are an important technique in all paradigms
  - In functional programming, loops are done with recursion
- We give a general technique, invariant programming, to program correct and efficient loops
  - Loops are often very difficult to get exactly right, and invariant programming is an excellent way to achieve this
  - We will see that this works for all paradigms, including functional programming and Java's object-oriented programming
- Each loop has one invariant
  - An invariant is a formula that is true at the beginning of each loop

44

## Doing loops with recursion…

- Let's take another look at SumDigitsR:

```
fun {SumDigitsR N}
    if (N==0) then 0
    else (N mod 10) + {SumDigitsR (N div 10)} end
end
```

- The recursive call and the condition together act like a loop: a calculation that is repeated to achieve a result
  - Each execution of the function body is one iteration of the loop
- We see that recursion can make a loop
  - Let's show a simpler example to understand better how it works!

45

## Naïve factorial function

- We can define factorial inductively:
  - 0! = 1
  - n! = n × (n-1)! when n>0
- We follow this definition to define a function:

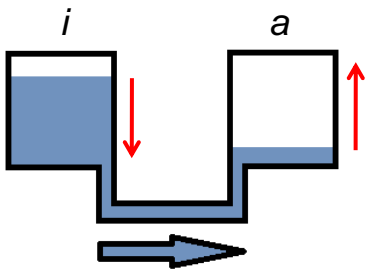
```
fun {Fact1 N}
    if N==0 then 1
    else N*{Fact1 N-1} end
end
```

- This looks ok, right? It's actually very bad!

46

# Better factorial function

- There is a much better way to define factorial!
- We start with an invariant, which is a logical formula that splits the work into two parts
  - *n! = i! × a*
- We do communicating vases
  - Here *n!* is constant and the two varying parts are *i!* and *a*
  - We start with *i=n* and *a=1*
  - We decrease *i* and increase *a*, while keeping the formula true
  - When *i=0* then *a* is the answer




47

# Better factorial function

- This gives us another factorial function:

```
fun {Fact2 I A}
    if I==0 then A
    else {Fact2 I-1 I*A)} end
end
{Browse {Fact2 N 1}}
```

- This one is much better than Fact1
  - Let's see why by looking at the execution of both

48

# Comparing Fact1 and Fact2

- 10 * {Fact1 10-1} ⇒
  10 * (9 * {Fact 9-1} ) ⇒
  10 * (9 * (8 * {Fact 8-1})) ⇒
  ...
  10 * (9 * (8 * (7 * (6 * (5 * (...(1 * {Fact 0})...) ⇒
  10 * (9 * (8 * (7 * (6 * (5 * (...(1 * 1)...) ⇒
  ...
  3628800

Each line does one computation step.
The stack size of Fact1 explodes!

- {Fact2 10-1 10*1} ⇒
  {Fact2 9-1 9*10} ⇒
  {Fact2 8-1 8*90} ⇒
  ...
  {Fact2 1-1 1*3628800}

? How can we make precise our intuition that Fact2 is better than Fact1? We need to introduce a formal semantics of the execution. → we will see this next week

49

# Fact2 is tail-recursive

- Tail recursion is when the recursive call is the last operation in the function body
- N * {Fact1 N-1}  % No tail recursion

  After Fact1 is done, we must come back for the multiply.
  Where is the multiplication stored? On a stack!

- {Fact2 I-1 I*A}  % Tail recursion
  The recursive call does not come back!
  All calculations are done *before* Fact2 is called.
  No stack is needed (memory usage is constant).

50

# Sum of digits using invariant programming

- Each recursive call handles one digit
- So we divide the initial number *n* into its digits:
  - $n = (d_{k-1}d_{k-2}\cdots d_2 d_1 d_0)$ *(where $d_i$ is a digit)*
- Let's call the sum of digits function *s(n)*
- Then we can split the work in two parts:
  - $s(n) = \underbrace{s(d_{k-1}d_{k-2}\cdots d_i)}_{s_i} + \underbrace{(d_{i-1} + d_{i-2} + \cdots + d_0)}_{a}$
    - $s_i$ is the work not yet done and *a* is the work already done
    - To keep the formula true, we set $i' = i+1$ and $a' = a+d_i$
    - When $i=k$ then $s_k=s(0)=0$ and therefore *a* is the answer

51

# Example execution

- Example with *n=314159*:

$s(n) = s(d_{k-1}d_{k-2}\cdots d_i) + (d_{i-1} + d_{i-2} + \cdots + d_0)$

  - s(314159) = s(314159) + 0
  - s(314159) = s(31415) + 9
  - s(314159) = s(3141) + 14
  - s(314159) = s(314) + 15
  - s(314159) = s(31) + 19
  - s(314159) = s(3) + 20
  - s(314159) = s(0) + 23 = 0 + 23 = 23

52

## Final SumDigits2 program

- S = $(d_{k-1}d_{k-2}\cdots d_i)$
  A = $(d_{i-1} + d_{i-2} + \cdots + d_0)$

```
fun {SumDigits2 S A}
    if S==0 then A
    else
        {SumDigits2 (S div 10) A+(S mod 10)}
    end
end
{Browse {SumDigits2 314159 0}}
```

53

## Invariant programming is best

- We have now programmed two problems
  - Factorial
  - Sum of digits
- For each problem we have defined two functions
  - First version based on a simple mathematical definition
  - Second version designed with invariant programming


- The second version has three interesting properties
  - It uses constant stack space, unlike the first version
  - It has two arguments: the growing one is called an accumulator
  - The recursive call is the last operation: it is tail-recursive

54

# Tail recursion = while loop

- A while loop in functional programming:

```
fun {While S}
   if {IsDone S} then S
   else {While {Transform S}} end /* tail recursion */
end
```

- A while loop in imperative programming:
  (i.e., in languages with multiple assignment like Java and C++)

```
state whileLoop(state s) {
   while (!isDone(s))
      s=transform(s); /* assignment */
   return s;
}
```

- In *both* cases, invariant programming is the right way to do loops

55

# Summary

56

## Summary

- Overview of course
- Practical organization
- Basic concepts
- Tips on Oz syntax
- Functional programming
- Invariant programming

- Next week:
  symbolic programming and formal semantics

57

# LINFO1104
# Concepts, paradigms, and semantics of programming languages

## Lecture 2 & 3

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

## Overview of lecture 2 & 3

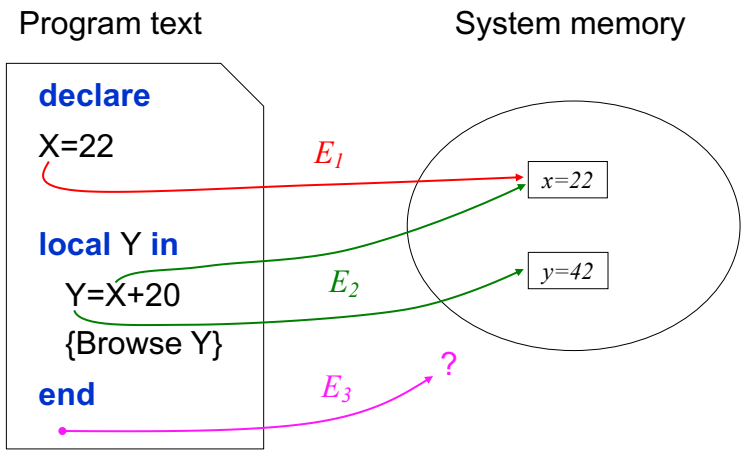
- Refresher of lecture 1
- Symbolic programming
  - Lists
  - Pattern matching
  - Trees
  - Tuples and records
- Formal semantics
  - Kernel language
  - Abstract machine
  - Proving correctness of programs
  - Semantic rules for kernel instructions
  - Semantics of procedures

2

# Refresher of lecture 1

3

# Program and memory…




4



4

## Environment

- Environments $E_1$, $E_2$, $E_3$
  - Function from identifiers to memory variables
  - A set of pairs $X \rightarrow x$
    - Identifier X, memory variable $x$
- Example environment $E_2$
  - $E_2$={X $\rightarrow$ $x$, Y $\rightarrow$ $y$}
  - $E_2$(X)=$x$
  - $E_2$(Y)=$y$

5

## An exercise on static scope

What does this program display?

```
local P Q in
   proc {P} {Browse 100} end
   proc {Q} {P} end
   local P in
      proc {P} {Browse 200} end
      {Q}
   end
end
```

6

# What is the scope of P?

```
local P Q in
   proc {P} {Browse 100} end
   proc {Q} {P} end
   local P in
      proc {P} {Browse 200} end
      {Q}
   end
end
```

7

# What is the scope of P?

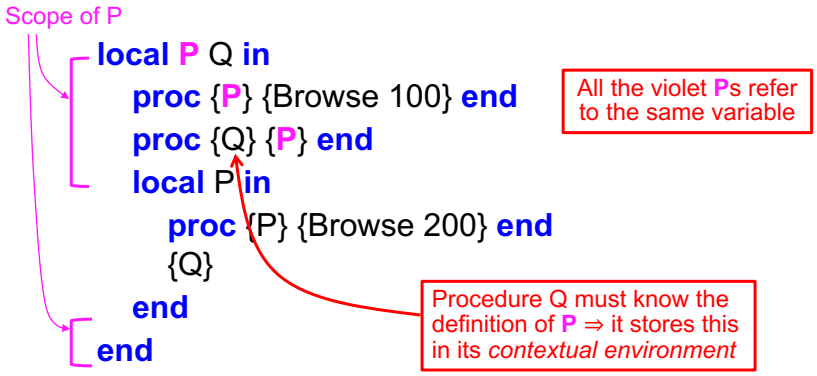
Scope of P

```
local P Q in
   proc {P} {Browse 100} end
   proc {Q} {P} end
   local P in
      proc {P} {Browse 200} end
      {Q}
   end
end
```

The P definition *inside* the scope

8

# Contextual environment of Q




9

# The contextual environment

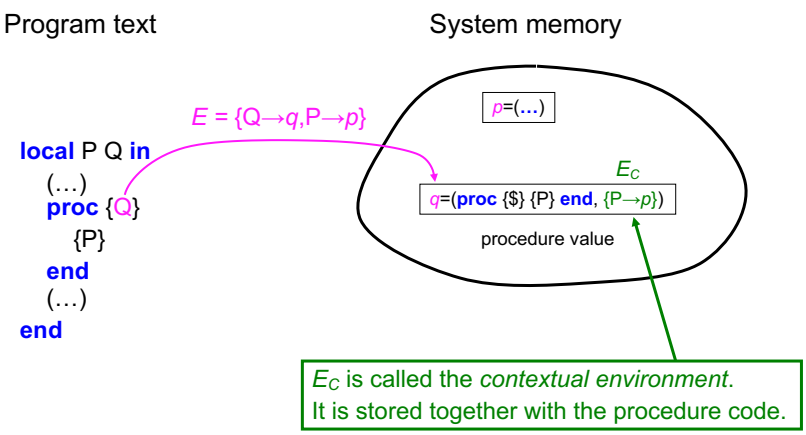
- The contextual environment of a function (or procedure) contains all the identifiers that are used *inside* the function but declared *outside* of the function

```
declare
A=1
proc {Inc X Y} Y=X+A end
```

- The contextual environment of Inc is $E_C$ = {A→*a*}
  - Where *a* is a variable in memory: *a=1*

10

# How procedure Q is stored in memory




11

# Procedure values

- A procedure value is stored in memory as a pair:

  *inc* = (**proc** {$ X Y} Y=X+A **end**, {A→*a*})

  Procedure code (without a name: '$' replaces the identifier)

  Contextual environment

- The variable *inc* is bound to the procedure value
  - Terminology: a procedure value is also called a *closure* or a *lexically scoped closure*, because it "closes" over the free identifiers when it is defined

12

# How Q is defined and called

- Recall the definition of Q:
  **proc** {Q} {P} **end**
  When Q is defined, an environment $E_c$ is created that contains P and $E_c$ is stored together with Q's code
  - $E_c$={P→*p*} is called the contextual environment of Q
- When Q is called, $E_c$ is used to get the right value *p*
  - This is guaranteed to always get the right value, even if there is another definition of P right next to the call of Q
- The identifiers in $E_c$ are the identifiers inside Q that are defined outside of Q
  - They are called the free identifiers of Q






# Free identifiers

- A free identifier of an instruction is an occurrence of an identifier inside the instruction that is declared outside the instruction
- The instruction:
  **local** Q **in**
  **proc** {Q A} {P A+1} **end**
  **end**
  has one free identifier:
  {P}
- The instruction:
  **local** Z **in** Z=X+Y **end**
  has two free identifiers:
  {X,Y}

# Lists

15

## Definition of a list

- A list is a recursive type: defined in terms of itself
  - Recursion is used both for computations and data!
  - We also use recursion for functions on lists
- A list is either an empty list or a pair of an element followed by another list
  - This definition is recursive because it defines lists in terms of lists. There is no infinite regress because the definition is used constructively to build larger lists from smaller lists.
- Let's introduce a formal notation

16

# Syntax definition of a list

- Using an EBNF grammar rule we write:

  <List T> ::= nil | T ‘|’ <List T>

- This defines the textual representation of a list
- EBNF = Extended Backus-Naur Form
  - Invented by John Backus and Peter Naur
  - <List T> represents a list of elements of type T
  - T represents one element of type T
- Be careful to distinguish between | and ‘|’ : the first is part of the grammar notation (it means “or”), and the second is part of the syntax being defined

17

# Some examples of lists

- According to the definition (if T is integer type):

  nil
  10 | nil
  10 | 11 | nil
  10 | 11 | 12 | nil
  10 | 11 | 12 | 13 | nil

18

# Type notation

- <Int> represents an integer; more precisely, it is the set of all syntactic representations of integers
- <List <Int>> represents the set of all syntactic representations of lists of integers
- T represents the set of all syntactic representations of values of type T; we say that T is a type variable
  - Do not confuse a type variable with an identifier or a variable in memory! Type variables exist only in grammar rules.



# Don't confuse a thing and its representation


René Magritte, *La trahison des images*, 1928-29, oil, Los Angeles County Museum of Art, Los Angeles.

- This is not a pipe.
  It is a digital display of a photograph of a painting of a pipe (thanks to Belgian surrealist René Magritte for pointing this out!).

1234

- This is not an integer.
  It is a digital display of a visual representation of an integer using numeric symbols in base 10.

# Representations for lists

- The EBNF rule gives one textual representation
  - <List <Int>> ⇒
    10 | <List <Int>> ⇒
    10 | 11 | <List <Int>> ⇒
    10 | 11 | 12 | <List <Int>> ⇒
    10 | 11 | 12 | nil

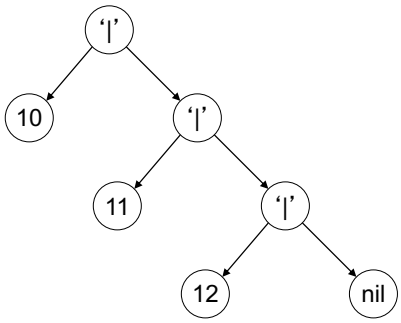
We repeatedly replace the left-hand side of the rule by a possible value, until no more can be replaced

- Oz allows another textual representation
  - Bracket notation: [10 11 12]
  - In memory, [10 11 12] is identical to 10 | 11 | 12 | nil
  - Different textual representations of the same thing are called syntactic sugar

21

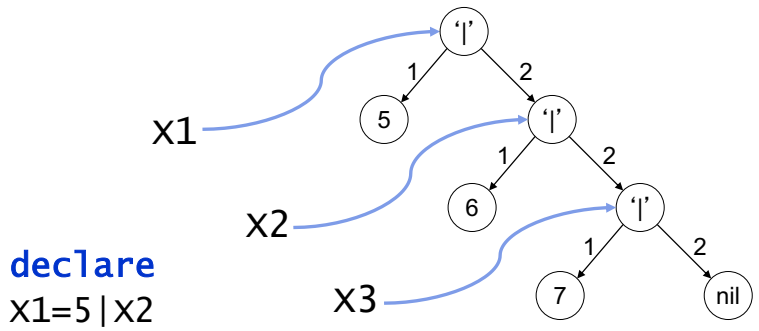
# Graphical representation of a list

- Graphical representations are very useful for reasoning
  - Humans have very powerful visual reasoning abilities
- We start from the leftmost pair, namely 10 | <List <Int>>
  - We draw three nodes with arrows between them
  - We then replace the node <List <Int>> as before
- This is an example of a more general structure called a tree




22

# Building a list incrementally




```
declare
X1=5|X2
X2=6|X3
X3=7|nil
```




23

# Computing with lists

- A non-empty list is a pair of head and tail
- Accessing the head:

  X.1
- Accessing the tail:

  X.2
- Comparing the list with nil:

  **if** X==nil **then** … **else** … **end**




24

## Head and tail functions

- We can define functions

```
fun {Head Xs}
   Xs.1
end

fun {Tail Xs}
   Xs.2
end
```




25

## Example with Head and Tail

- {Head [a b c]}
  returns a
- {Tail [a b c]}
  returns [b c]
- {Head {Tail {Tail [a b c]}}}
  returns c

- Draw the graphical picture of [a b c]!




26

# Functions on lists

27

## Functions that create lists

- Let us now define a function that outputs a list
  - We will use both pattern matching and recursion, as before, but this time the output will also be a list
  - We will define the Sum function to compute the sum of elements of a list
  - We give first the naïve version and then the smart version (based on invariants)

28

# Sum of list elements

- We are given a list of integers
- We would like to calculate their sum
  - We will define the function "Sum"
- Inductive definition following the list structure
  - Sum of an empty list: 0
  - Sum of a non-empty list L: {Head L} + {Sum {Tail L}}




29

# Sum of list elements (naïve method)

```
fun {Sum L}
   if L==nil then
      0
   else
      {Head L} + {Sum {Tail L}}
   end
end
```




30

# Sum of list elements (with accumulator)

```
fun {Sum2 L A}
   if L==nil then
      A
   else
      {Sum2 {Tail L} A+{Head L}}
   end
end
```

What is the invariant?





31

# Another example: Nth function

- Define the function {Nth L N} which returns the nth element of L
- The type of Nth is:
  <fun {$ <List T> <Int>}:<T>>
- Reasoning:
  - If N==1 then the result is {Head L}
  - If N>1 then the result is {Nth {Tail L} N-1}





32

## The Nth function

- The complete definition:

```
fun {Nth L N}
   if N==1 then {Head L}
   elseif N>1 then
      {Nth {Tail L} N-1}
   end
end
```

- What happens if the nth element does not exist?

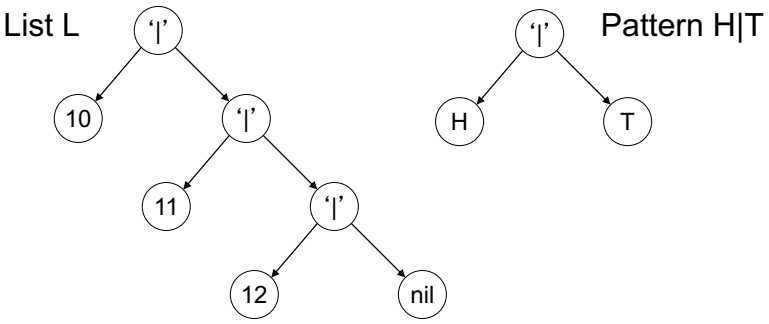



33

## Pattern matching

34

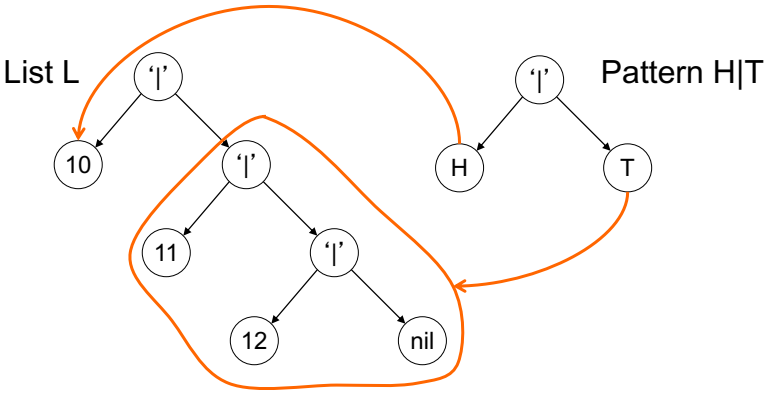
# Pattern matching




- **case** L **of** H|T **then** ... **else** ... **end**

35

# Pattern matching




- **case** L **of** H|T **then** ... **else** ... **end**
- H=10, T=11|12|nil

36

# Sum with pattern matching

```
fun {Sum L}
   case L
   of nil then 0
   [] H|T then H+{Sum T}
   end
end
```




37

# Sum with pattern matching

```
fun {Sum L}
   case L
   of nil then 0
   [] H|T then H+{Sum T}
   end
end
```

*A clause*

- "nil" is the *pattern* of the clause




38

# Sum with pattern matching

```
fun {Sum L}
   case L
   of nil then 0
   [] H|T then H+{Sum T}
   end
end
```

*A clause*

- "H|T" is the *pattern* of the clause




39

# Pattern matching

- The first clause uses **of**, the others use **[]**
- Clauses are tried in their textual order
- A clause matches if its pattern matches
- A pattern matches if its label and its arguments match
  - The identifiers in the pattern are assigned to their corresponding values in the input
- The first matching clause is executed, following clauses are ignored




40

# Kernel language introduction

41

## The kernel language

- The kernel language is the first part of the formal semantics of a programming language
  - The second part is the *abstract machine* which we will see later on
- Remember in lecture 1, we explained that each programming paradigm has a simple core language called its kernel language
  - We now introduce the kernel language of functional programming
- All programs in functional programming can be translated into the kernel language
  - All intermediate results of calculations are visible ← Kernel principle
  - All functions become procedures with one extra argument
  - Nested function calls are unnested by introducing new identifiers

42

# Length of a list

```
fun {Length Xs N}
   case Xs
   of nil then N
   [] X|Xr then {Length Xr N+1}
   end
end
```




43

# Length of a list translated into kernel language

- The instruction **case** (with one pattern) is part of the kernel language:

```
proc {Length Xs N R}
   case Xs
   of nil then R=N
   else
      case Xs
      of X|Xr then
         local N1 in
            N1=N+1
            {Length Xr N1 R}
         end
      else
         raise typeError end /* type error: see later in the course! */
      end
   end
end
```




44

# A function is a procedure with one extra argument

- The kernel language does not need functions
  - It's enough to have procedures
  - Factored design: each concept occurs only once
- A function is translated as a procedure with one extra argument, which gives the function's result
- N={Length L Z}
  is equivalent to:
  {Length L Z N}

45

# Translating to kernel language

- All practical programs can be translated into kernel language
- How to translate:
  - Only kernel language instructions can be used
  - The consequence is that all « hidden » variables become visible
    - Functions become procedures with one extra argument
    - Nested expressions become sequences, with extra local identifiers
    - Each pattern has its own case statement
  - The kernel language is a subset of Oz!
    - It can be executed in Mozart
- Consequences:
  - Kernel programs are longer
  - It is easy to see when programs are tail-recursive
  - It is easy to see exactly how programs execute

46

# Kernel language of the functional paradigm (so far)

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | **proc** {<x> <x>$_1$ … <x>$_n$} <s> **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**

This is almost complete!

- <v> ::= <number> | <list> | **...**
- <number> ::= <int> | <float>
- <list>, <p> ::= nil | <x> | <x> '|' <list>

still something missing

47

# Trees

48

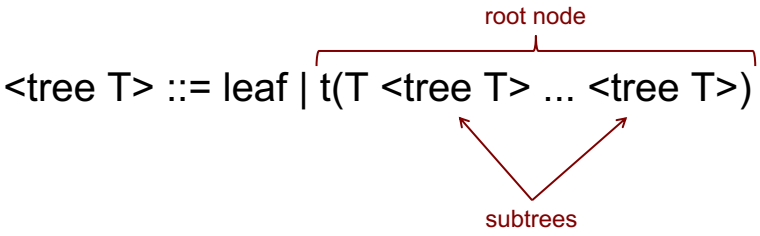
# Trees

- Trees are the second most important data structure in computing, next to lists
  - Trees are extremely useful for efficiently organizing information and performing many kinds of calculations
- Trees illustrate well goal-oriented programming
  - Many tree data structures are based on a global property, that must be maintained during the calculation
- In this lesson we will define trees and use them to store and look up information
  - We will define ordered binary trees and algorithms to add information, look up information, and remove information

49

# Trees

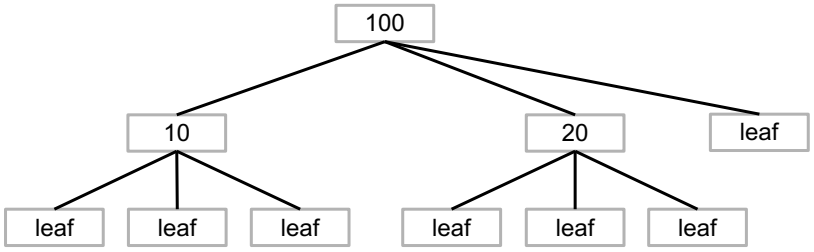
- A tree is a recursive structure: it is either an empty tree (called a leaf) or an element and a set of trees




50

# Example tree

- **declare**
  T=t(100 t(10 leaf leaf leaf) t(20 leaf leaf leaf) leaf)




51

# Trees compared to lists

- A tree is a recursive structure: it is either an empty tree (called a leaf) or an element and a set of trees

  <tree T> ::= leaf | t(T <tree T> ... <tree T>)

  <list T> ::= nil | '|'(T <list T>)

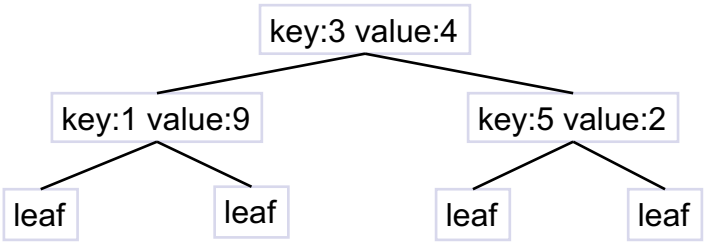
Notice the similarity with lists!

52

# Ordered binary tree (1)

- <obtree T> ::= leaf
  | tree(key:T value:T left:<obtree T> right:<obtree T>)

- Binary: each non-leaf tree has two subtrees (named left and right)
- Ordered: for each tree (including all subtrees):
  all keys in the left subtree < key of the root
  key of the root < all keys in the right subtree

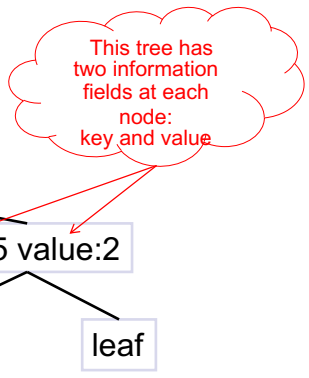



53

# Ordered binary tree (2)

- <obtree T> ::= leaf
  | tree(key:T value:T left:<obtree T> right:<obtree T>)

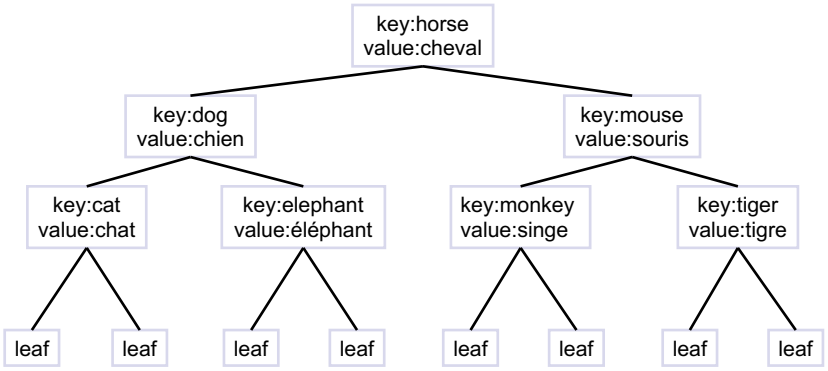
- Binary: each non-leaf tree has two subtrees (named left and right)
- Ordered: for each tree (including all subtrees)
  all keys in the left subtree < key of the root
  key of the root < all keys in the right subtree




54

# Ordered binary tree (3)

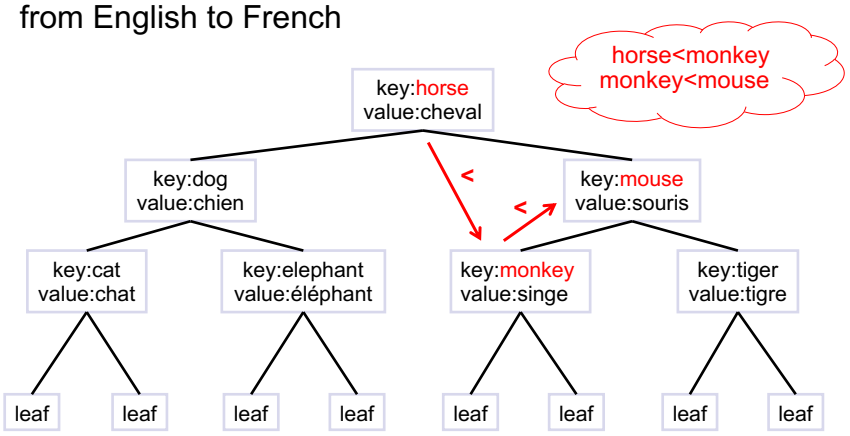
- This ordered binary tree is a translation dictionary from English to French




55

# Ordered binary tree (4)

- This ordered binary tree is a translation dictionary from English to French



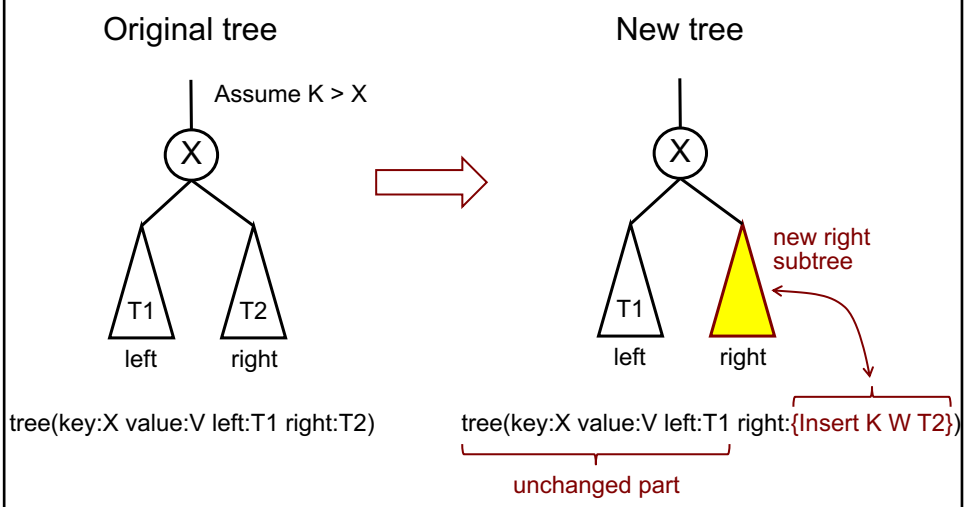

56

# Search tree

- Search tree: A tree that is used to organize information, and with which we can perform various operations such as looking up, inserting, and deleting information
- Let's define these three operations:
  - {Lookup K T}: returns the value V corresponding to key K
  - {Insert K W T}: returns a new tree with added (K,W)
  - {Delete K T}: returns a new tree that does not contain K

57

# Inserting a new key/value pair




58

# Efficiency of Lookup

- How efficient is the Lookup function?
  - If there are *n* words in the tree, and each node's subtrees are approximately equal in size (we say the tree is balanced), then the average lookup time is proportional to $\log_2 n$
  - Tree lookup is much more efficient than list lookup: if for 1000 words the average time is 10, then for 1000000 words this will increase to 20 (instead of being multiplied by 1000)
- If the tree is not balanced, say all the right subtrees are very small, then the time will be much larger
  - In the worst case, the tree will look like a list
- How can we arrange for the tree to be balanced?
  - There exist algorithms for balancing an unbalanced tree, but if we insert words randomly, then we can show that the tree will be approximately balanced, good enough to achieve logarithmic time

59

# Looking up information

- There are four possibilities:
- K is not found
- K is found
- K might be in the left subtree
- K might be in the right subtree

```
fun {Lookup K T}
   case T
   of leaf then notfound
   [] tree(key:Y value:V T1 T2) andthen K==Y then
         found(V)
   [] tree(key:Y value:V T1 T2) andthen K<Y then
         {Lookup K T1}
   [] tree(key:Y value:V T1 T2) andthen K>Y then
         {Lookup K T2}
   end
end
```

60

# Inserting information

- There are four possibilities:
- (K,W) replaces a leaf node
- (K,W) replaces an existing node
- (K,W) is inserted in the left subtree
- (K,W) is inserted in the right subtree

```
fun {Insert K W T}
   case T
   of leaf then tree(key:K value:W leaf leaf)
   [] tree(key:Y value:V T1 T2) andthen K==Y then
         tree(key:K value:W T1 T2)
   [] tree(key:Y value:V T1 T2) andthen K<Y then
         tree(key:Y value:V {Insert K W T1} T2)
   [] tree(key:Y value:V T1 T2) andthen K>Y then
         tree(key:Y value:V T1 {Insert K W T2})
   end
end
```

61

# Deleting information

- There are four possibilities:
- (K,_) is not in the tree
- (K,_) is removed immediately
- (K,_) is removed from the left subtree
- (K,_) is removed from the right subtree
- Right?

```
fun {Delete K T}
   case T
   of leaf then leaf
   [] tree(key:Y value:W T1 T2) andthen K==Y then
         leaf
   [] tree(key:Y value:W T1 T2) andthen K<Y then
         tree(key:Y value:W {Delete K T1} T2)
   [] tree(key:Y value:W T1 T2) andthen K>Y then
         tree(key:Y value:W T1 {Delete K T2})
   end
end
```

62

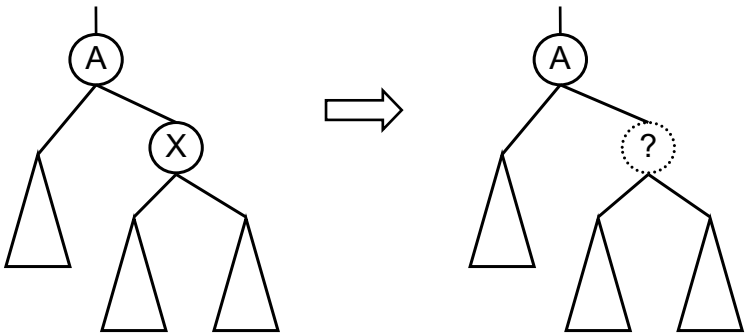
# Deleting information

- There are four possibilities:
- (K,_) is not in the tree
- (K,_) is removed immediately
- (K,_) is removed from the left subtree
- (K,_) is removed from the right subtree
- Right? WRONG!

```
fun {Delete K T}
   case T
   of leaf then leaf
   [] tree(key:Y value:W T1 T2) andthen K==Y then
         leaf
   [] tree(key:Y value:W T1 T2) andthen K<Y then
         tree(key:Y value:W {Delete K T1} T2)
   [] tree(key:Y value:W T1 T2) andthen K>Y then
         tree(key:Y value:W T1 {Delete K T2})
   end
end
```

63

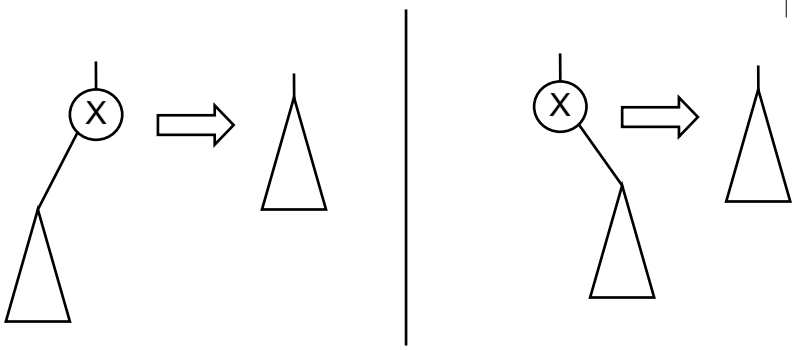
# Deleting an element from an ordered binary tree




The problem is to repair the tree after X disappears

64

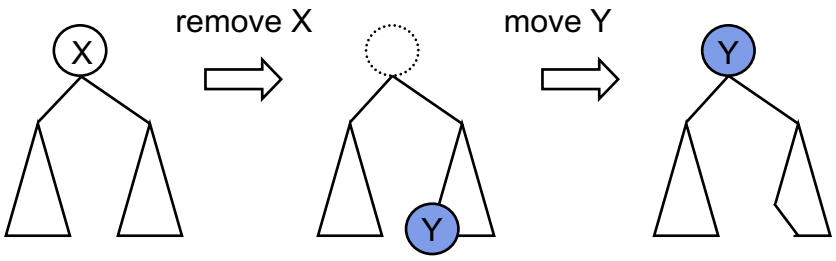
# Deleting the root when one subtree is empty




It's easy when one of the subtrees is empty: just replace the tree by the other subtree

65

# Deleting the root when both subtrees are not empty

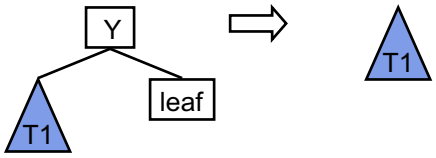


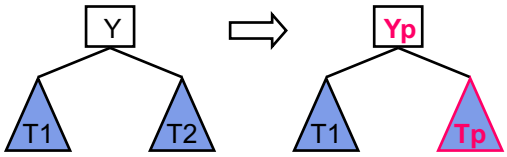

The idea is to fill the "hole" that appears after X is removed. We can put there the smallest element in the right subtree, namely Y.

66

# Deleting the root

- To remove the root Y, there are two possibilities:
- One subtree is a leaf. Just return the other.
- Neither subtree is a leaf. Remove an element from one of its subtrees.

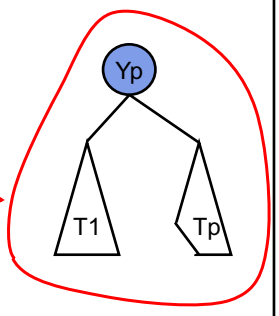



67

# We need a new function: RemoveSmallest

```
fun {Delete K T}
   case T
   of leaf then leaf
   [] tree(key:X value:V left:T1 right:T2) andthen K==X then
         case {RemoveSmallest T2}
         of none then T1
         [] triple(Tp Yp Vp) then
              tree(key:Yp value:Vp left:T1 right:Tp)
         end
   [] … end
end
```




- RemoveSmallest takes a tree and returns three values:
  - The new subtree Tp without the smallest element
  - The smallest element's key Yp
  - The smallest element's value Vp
- With these three values we can build the new tree where Yp is the root and Tp is the new right subtree

68

# Recursive definition of RemoveSmallest

```
fun {RemoveSmallest T}
   case T
   of leaf then none
   [] tree(key:X value:V left:T1 right:T2) then
      case {RemoveSmallest T1}
      of none then triple(T2 X V)
      [] triple(Tp Xp Vp) then
         triple(tree(key:X value:V left:Tp right:T2) Xp Vp)
      end
   end
end
```

To understand this definition, draw diagrams with trees!

- RemoveSmallest takes a tree T and returns:
  - The atom none when T is empty
  - The record triple(Tp Xp Vp) when T is not empty

69

# Delete operation is complex

- Why is the delete operation so complex?
- It is because the tree satisfies a global condition, namely it is ordered
- The delete operation has to work to keep this condition true
- Many tree algorithms depend on global conditions and must work to keep the conditions true
- The interesting thing about a global condition is that it gives the tree **a spark of life**: the tree behaves a bit like it is alive (« goal-oriented behavior»)
  - Living organisms have goal-oriented behavior

70

## Goal-oriented programming

- Many tree algorithms depend on global properties and most of the work they do is in maintaining these properties
  - The ordered binary tree satisfies a global ordering condition. The insert and delete operations must maintain this condition. This is easy for insert, but harder for delete.
- Goal-oriented programming is widely used in artificial intelligence algorithms
  - It can give unexpected results as the algorithm does its thing to maintain the global property.
  - Goal-oriented behavior is characteristic of living organisms. So defining algorithms that are goal-oriented gives them a spark of life!

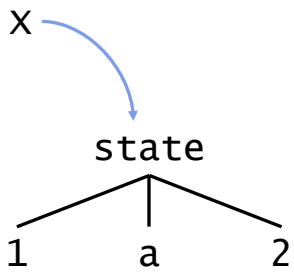
71

## Tuples and records

72

# Tuples

X=state(1 a 2)

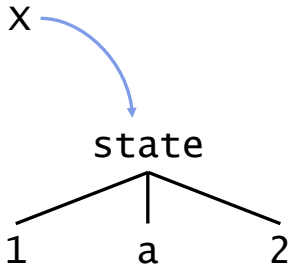



- A tuple allows grouping several values together
  - For example: 1, a, 2
  - The position is meaningful: first, second, third!
- A tuple has a label
  - For example: `state`

73

# Operations on tuples

X=state(1 a 2)

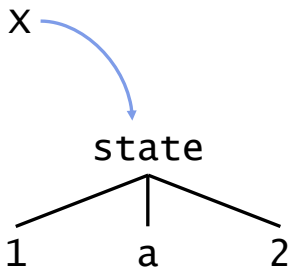



- `{Label X}` returns *the label* of tuple `X`
  - For example: `state`
  - The label is a constant, called an atom
- `{Width X}` returns *the width* (number of fields)
  - For example: 3
  - Always a positive integer or zero

74

# Accessing fields (“.” operation)

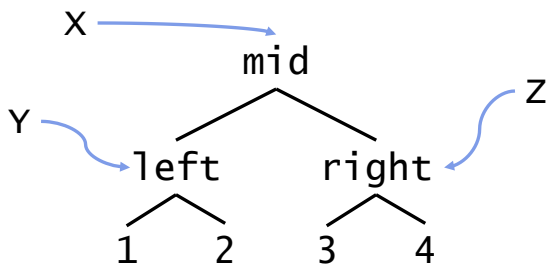
X=state(1 a 2)




- Fields are numbered from `1` up to `{Width X}`
- `X.N` returns the nth field of tuple X:
  - `X.1` returns 1
  - `X.3` returns 2
- In the expression `X.N`, N is called the field name or “feature”

75

# Building a tree




- A tree can be built with tuples:

```
declare
Y=left(1 2) Z=right(3 4)
X=mid(Y Z)
```

76

## Testing equality (==)

- Equality testing with a number or atom
  - Easy: the number or atom must be the same
- Equality testing of trees
  - Also easy: the two trees must have the same root tuples and the same subtrees in corresponding fields
  - Careful when the tree has a cycle!
    - Comparison with == works, but naïve programs may loop
    - Advice: avoid this kind of tree

77

## Tuples summary

- Tuple
  - Label
  - Width
  - Field
  - Field name, feature
- Accessing fields with "." operation
- Build trees with tuples
- Pattern matching with tuples
- Comparing tuples with "=="

78

## A list is a tuple

- The list H|T is actually a tuple '|' (H T)
- Principle of simplicity in the kernel language: instead of two concepts (tuples and lists), only one concept is needed (tuple)
- Because of their usefulness, lists have a syntactic sugar
  - It is purely for programmer comfort, it makes no difference in the kernel language

79

## Syntax of lists as tuples

- A list is a special case of a tuple
- Prefix syntax (put the label '|' in front)
  - Nil
    '|'(5 nil)
    '|'(5 '|'(6 nil))
    '|'(5 '|'(6 '|'(7 nil)))
- Prefix syntax with field names
  - Nil
    '|'(1:5 2:nil)
    '|'(1:5 2: '|'(1:6 2:nil))
    '|'(1:5 2: '|'(1:6 2: '|'(1:7 2:nil)))


80

80

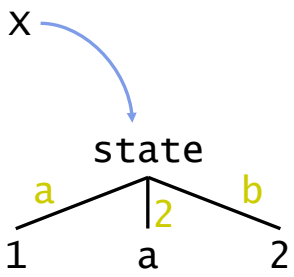
# Records

- A record is a generalization of a tuple
  - Field names can be atoms (i.e., constants)
  - Field names can be any integer
    - Does not have to start with 1
    - Does not have to be consecutive
- A record also has a label and a width

81

# Records

```
X=state(a:1 2:a b:2)
```




- The position of a field is no longer meaningful
  - Instead, it is the field name that is meaningful
- Accessing fields is done the same as for tuples
  - `X.a=1`

82

## Record operations

- Label and width operations:
  - {Label X}=state
  - {Width X}=3
- Equality test:
  - X==state(a:1 b:2 2:a)
- New operation: arity
  - Returns a list of field names
  - {Arity X}=[2 a b] (in lexicographic order)
  - Arity also works for tuples and lists!

83

## A tuple is a record

- The record:

  ```
  X = state(1:a 2:b 3:c)
  ```

  is the same as the tuple:

  ```
  X = state(a b c)
  ```

- In a tuple, all fields are numbered consecutively from 1

- What happens if we write:

  ```
  X = state(a 2:b 3:c)
  ```

  or

  ```
  X = state(2:b 3:c a)
  ```

- In a record, all unnamed fields are numbered consecutively starting with 1

84

# A list is a tuple and a tuple is a record ⇒ many list syntaxes

- The list syntax

  ```
  X1=5|6|7|nil
  ```

  is a short-cut for

  ```
  X1=5|(6|(7|nil))
  ```

  which is a short-cut for

  ```
  X1='|'(5 '|'(6 '|'(7 nil)))
  ```

  which is a short-cut for

  ```
  X1='|'(1:5 2:'|'(1:6 2:'|'(1:7 2:nil)))
  ```

- The shortest syntax (the 'nil' is implied!)

  ```
  X1=[5 6 7]
  ```






# The kernel language has only records

- In the kernel language there are only records
  - An atom is a record whose width is 0
  - A tuple is a record whose field names are numbered consecutively starting from 1
    - If this condition is not satisfied, the data structure is still a record but it is no longer a tuple
  - A list is built with tuples nil and '|' (X Y)
- This keeps the kernel language simple
  - It has just one data structure

# Kernel language with records

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | **proc** {<x> <x>$_1$ … <x>$_n$} <s> **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <record> | …
- <number> ::= <int> | <float>
- <record>, <p> ::= <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

Records replace lists

87

# Exercises

- Which of these records are tuples?
  - A=a(1:a 2:b 3:c)
  - B=a(1:a 2:b 4:c)
  - C=a(0:a 1:b 2:c)
  - D=a(1:a 2:b 3:c d)
  - E=a(a 2:b 3:c 4:d)
  - F=a(2:b 3:c 4:d a)
  - G=a(1:a 2:b 3:c foo:d)
  - H= '|' (1:a 2:' |' (1:b 2:nil))
  - I= '|' (1:a 2:' |' (1:b 3:nil))

88

# Introduction to formal semantics

89

## Why do we need semantics?

- If you do not understand something, then you do not master it – it masters you!
  - If you know nothing about how a car works, then a car mechanic can charge you whatever he wants
  - If you do not understand how government works, then you cannot vote wisely and the government becomes a tyranny
- The same holds true for programming
  - To write correct programs and to understand other people's programs, you have to understand the language deeply
  - All software developers should have this level of understanding
  - This understanding comes with the formal semantics

90

# What is the semantics of a language?

- The semantics of a programming language is a fully precise explanation of how programs execute
  - With it we can reason about program design and correctness
- We give the semantics for all paradigms of this course
  - We start by giving the semantics of functional programming
- Before taking the plunge, let's take a step back and talk about semantics in general

91

# Different approaches to define language semantics

- Four general approaches have been invented:
  - Operational semantics: Explains a program in terms of its execution on a rigorously defined abstract machine
    - This works for all paradigms!

This course

  - Axiomatic semantics: Explains a program as an implication: if certain properties hold before the execution, then some other properties will hold after the execution
    - « If the precondition holds before, then the postcondition will hold after » as shown in LEPL1402
    - This works well for imperative paradigms (like object-oriented programming as in Java)

LEPL1402

  - Denotational semantics: Explains a program as a function over an abstract domain, which simplifies certain kinds of mathematical analysis of the program
    - This works well for functional programming languages
  - Logical semantics: Explains a program as a logical model of a set of logical axioms, so program execution is deduction: the result of a program is a true property derived from the axioms
    - This works well for logic programming languages such as Prolog and constraint programming
- We will focus on operational semantics

92

# Operational semantics

- The operational semantics has two parts
  - Kernel language: first translate the program into the kernel language
  - Abstract machine: then execute the program on the abstract machine
- We will introduce the operational semantics in five parts
  1. The full kernel language for functional programming
  2. Executing an example program on the abstract machine
  3. Defining the abstract machine and its semantic rules
  4. Proving the correctness of an example program
  5. Procedure definition and call are special because they are the foundation of data abstraction. We define the semantic rules of procedure definition and call.

93

# Semantics 1: Full kernel language

94

# Kernel language of functional programming

- We have seen all concepts of functional programming
  - Now we can define its full kernel language
- We will use this kernel language to understand exactly what a functional program does
  - We have used it to see why list functions are tail-recursive
  - We will use it to prove correctness of programs
- Each time we introduce a new paradigm in the course we will define its kernel language
  - Each extends the functional kernel language with a new concept

95

# The functional kernel language (what we saw before)

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | **proc** {<x> <x>$_1$ … <x>$_n$} <s> **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**

This is the kernel language with lists and procedure statements

- <v> ::= <number> | <list> | ...
- <number> ::= <int> | <float>
- <list>, <p> ::= nil | <x> | <x> '|' <list>

Still incomplete

96

# The functional kernel language

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | **proc** {<x> <x>$_1$ … <x>$_n$} <s> **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <list> | ...
- <number> ::= <int> | <float>
- <list>, <p> ::= nil | <x> | <x> '|' <list>

This is what we have seen so far; it needs *two changes* to become the full kernel language of the functional paradigm

1. Procedure declarations (should be values)

2. Records instead of lists (records subsume lists)

97

# The functional kernel language (procedure values)

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | ~~**proc** {<x> <x>$_1$ … <x>$_n$} <s> **end**~~
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <procedure> | <list> | ...
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <list>, <p> ::= nil | <x> | <x> '|' <list>

1. Procedures are values in memory (like numbers and lists)

This is called an "anonymous procedure". The procedure name is replaced by a placeholder "$".

98

# The functional kernel language (records)

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <procedure> ~~| <list>~~ | <record>
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <record>, <p> ::= <lit> | <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

2. Records subsume lists

99

# The functional kernel language (complete)

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
- <v> ::= <number> | <procedure> | <record>
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <record>, <p> ::= <lit> | <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

Procedure values and records are important basic types. They allow to define data abstractions including all of object-oriented programming.

100

# Semantics 2: Executing with the abstract machine

101

## Executing a program with the abstract machine

- We execute the program using the semantics by following two steps
- First, we translate the program into kernel language
  - We use the kernel language of functional programming
  - All programs can be translated into kernel language
- Second, we execute the translated program on the abstract machine
  - The abstract machine is a simplified computer with a precise mathematical definition

→ Let's see an example execution

102

# The example program in kernel language

```
local X in
   local B in
      B=true
      if B then X=1 else skip end
   end
end
```

103

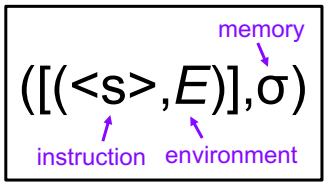
# Start of the execution: the initial execution state

```
([(local X in
     local B in
       B=true
       if B then X=1 else skip end
     end
   end, {})],
{})
```

instruction stack […]

empty environment

empty memory




Execution state

- The initial execution state has an empty memory {} and an empty environment {}
- We start execution with ***local*** *X* ***in*** <*s*> ***end***

104

# The *local X in ... end* instruction

```
([(local B in
      B=true
      if B then X=1 else skip end
   end,
   {X → x})],
 {x})
```

- We create a new variable *x* so the memory becomes {*x*}
- We create a new environment {X → *x*} so that X can refer to the new variable *x*

105

# The *local B in ... end* instruction

```
([((B=true
      if B then X=1 else skip end),
   {B → b, X → x})],
 {b,x})
```

- We create a new variable *b* in memory
- We put the inner instruction on the stack and add B→*b* to its environment, giving {B → *b*, X → *x*}

106

## The sequential composition instruction

([(B=**true**,{B → *b*, X → *x*}),
(**if** B **then** X=1
**else skip end**,{B → *b*, X → *x*})],
{*b*,*x*})

- We split the sequential composition into its two parts
  - B=**true** *and* **if** B **then** X=1 **else skip end**
- We put the two instructions on the stack
- Each instruction gets the same environment

107

## The *B=true* instruction

([(**if** B **then** X=1
**else skip end**,{B → *b*, X → *x*})],
{*b*=**true**, *x*})

- We bind variable *b* to **true** in memory

108

## The conditional instruction

([(X=1,{B → *b*, X → *x*})],
{*b*=**true**, *x*})

- We read the value of B
- Since B is **true**, it puts the instruction after **then** on the stack
- If B is **false**, it will put the instruction after **else** on the stack
- If B has any other value, then the conditional raises an error

- (Note: If B is unbound then the execution of the semantic stack stops until B becomes bound – this can only happen in another semantic stack, i.e., with concurrency, as we will see)

109

## The *X=1* instruction

([],
{*b*=**true**, *x*=1})

- We bind *x* to 1 in memory
- Execution stops because the stack is empty

110

## Semantic rules we have seen

- This example has shown us the execution of four instructions:
  - **local** <x> **in** <s> **end** (variable creation)
  - $\langle s \rangle_1$ $\langle s \rangle_2$ (sequential composition)
  - **if** <x> **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end** (conditional)
  - <x>=<v> (assignment)
- We will define the semantic rules corresponding to these instructions

111

## Semantics 3: The abstract machine

112

# All abstract machine concepts

- Single-assignment memory $\sigma = \{x_1=10, x_2, x_3=20\}$
  - Variables and the values they are bound to
- Environment $E = \{X \rightarrow x, Y \rightarrow y\}$
  - Link between identifiers and variables in memory
- Semantic instruction (<s>,$E$)
  - An instruction with its environment
- Semantic stack ST = [(<s>$_1$,$E_1$), …, (<s>$_n$,$E_n$)]
  - A stack of semantic instructions
- Execution state (ST,$\sigma$)
  - A pair of a semantic stack and a memory
- Execution $(ST_1,\sigma_1) \rightarrow (ST_2,\sigma_2) \rightarrow (ST_3,\sigma_3) \rightarrow \ldots$
  - A sequence of execution states

113

# Abstract machine execution algorithm

- **procedure** execute(<s>)
  **var** ST, σ, SI;
  **begin**
  ST:=[(<s>,{})]; /* Initial semantic stack: one instruction, empty env. */
  σ:={}; /* Initial memory: empty (no variables) */
  **while** (ST≠{}) **do**
  SI:=top(ST); /* Get topmost element of semantic stack */
  (ST,σ):=rule(SI, (ST,σ)); /* Execute SI according to its rule */
  **end**
  **end**

  each kernel instruction has a rule

- While the semantic stack is nonempty, get the instruction at the top of the semantic stack, and execute it according to its semantic rule
- Each instruction of the kernel language has a rule that defines its execution
- (Note: When we introduce concurrency, we will extend this algorithm to run with more than one semantic stack)

114

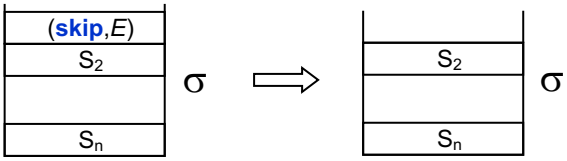
# Semantic rules for kernel language instructions

- For each instruction in the kernel language, we will define its rule in the abstract machine
- Each instruction takes one execution state as input and returns one execution state
  - Execution state = semantic stack ST + memory σ
- Let's look at three instructions in detail:
  - **skip**
  - <s>$_1$ <s>$_2$ (sequential composition)
  - **local** <x> **in** <s> **end**
- We will see the others in less detail. You can learn about them in the exercises and in the book.

115

# skip

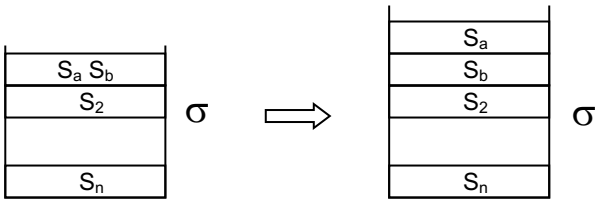
- The simplest instruction
- It does nothing at all!
- Input state: ([(**skip**,*E*), $S_2$, …, $S_n$], σ)
- Output state: ([$S_2$, …, $S_n$], σ)
- That's all




116

## (<s>$_1$ <s>$_2$) (sequential composition)

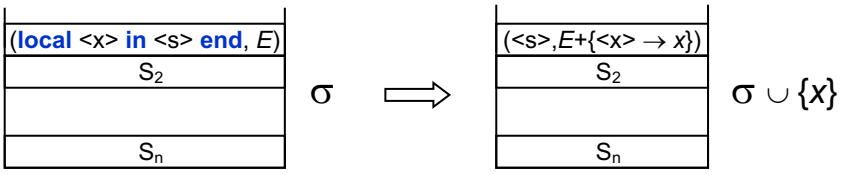
- Almost as simple as **skip**
- The instruction removes the top of the stack and adds two new elements
- Input state: ([($S_a$ $S_b$), $S_2$, …, $S_n$], σ)
- Output state: ([$S_a$, $S_b$, $S_2$, …, $S_n$], σ)




117

## local <x> in <s> end

- Create a fresh new variable x in memory σ
- Add the pair {X → *x*} to the environment *E* (using adjunction operation)




118

## Some other instructions

- <x>=<v> (value creation + assignment)
  - Note: when <v> is a procedure, you have to create the contextual environment
- **if** <x> **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end** (conditional)
  - Note: if <x> is unbound, the instruction will wait ("block") until <x> is bound to a value
  - The activation condition: "<x> is bound to a value"
- **case** <x> **of** <p> **then** $\langle s \rangle_1$ **else** $\langle s \rangle_2$ **end**
  - Note: **case** statements with more patterns are built by combining several kernel instructions
- {<x> $\langle y \rangle_1$ ... $\langle y \rangle_n$}
  - Note: since procedure definition and procedure call are the foundation of data abstraction, we will take a special look!

119

## Semantics 4: Proving correctness with the semantics

120

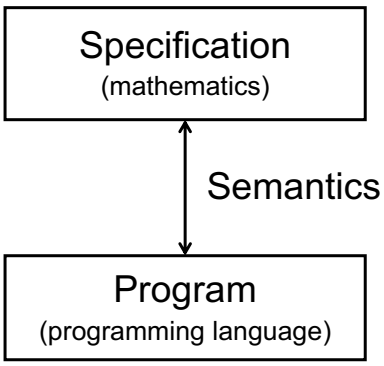
# When is a program correct?

- "A program is correct when it does what we want"
  - How can we be sure?
- We need to make precise what we want it to do:
  - We introduce the concept of specification
- We need to prove that the program satisfies the specification, when it executes according to the semantics

121

# The three pillars

- The specification: what we want
- The program: what we have
- The semantics connects these two: proving that what we have executes according to what we want




122

# Example: correctness of factorial

- The specification of {Fact N} (mathematics)

  0! = 1

  $n!$ = $n \times ((n\text{-}1)!)$ when $n>0$

- The program (programming language)

```
fun {Fact N}
    if N==0 then 1 else N*{Fact N-1} end
end
```

- The semantics connects the two
  - Executing R={Fact N} gives the result *r=n!*

123

# Mathematical induction

- To make this proof for a recursive function we need to use mathematical induction
  - A recursive function calculates on a recursive data structure, which has a base case and a general case
  - We first show the correctness for the base case
  - We then show that if the program is correct for a general case, it is correct for the next case
- For integers, the base case is usually 0 or 1, and the general case *n-1* leads to the next case *n*
- For lists, the base case is usually nil or a small list, and the general case T leads to the next case H|T

124

# The inductive proof

- We must show that {Fact N} calculates *n!* for all $n \geq 0$
- Base case: *n=0*
  - The specification says: *0!=1*
  - The execution of {Fact 0}, *using the semantics*, gives {Fact 0}=1
    - *It's correct!*
- General case: *(n-1)* → *n*
  - The specification says: $n! = n \times (n-1)!$
  - The execution of {Fact N}, *using the semantics*, gives {Fact N} = $n \times$ {Fact N-1}
    - We assume that {Fact N-1}=*(n-1)!*
    - We assume that the language correctly implements multiplication
    - Therefore: {Fact N} = $n \times$ {Fact N-1} = $n \times (n-1)! = n!$
    - *It's correct!*
- Now we just need to understand the magic words "*using the semantics*"!

125

# How to execute a program *using the semantics*

- We execute the program using the semantics by following two steps
- First, we translate the program into kernel language
  - The kernel language is a simple language that has all essential concepts
  - All programs can be translated into kernel language
  - → We translate the definition of Fact into kernel language
- Second, we execute the translated program on the abstract machine
  - The abstract machine is a simplified computer with a precise definition
  - → We execute {Fact 0 R} and {Fact N R} on the abstract machine

126

# Executing Fact using the semantics

- We need to execute both {Fact 0} and {Fact N} using the semantics
- First we translate the definition of Fact into kernel language:

```
proc {Fact N R}
   local B in
      B=(N==0)
      if B then R=1
      else local N1 R1 in
         N1=N-1
         {Fact N1 R1}
         R=N*R1
      end
   end
end
```

127

# Execution of {Fact 0} (1)

- Let's first look at the function call {Fact 0}
- We execute the procedure call {Fact N R} where N=0
- We need a memory σ and an environment *E*:

  σ = {*fact*=(**proc** {$ N R} … **end**,{Fact→*fact* }), *n*=0, *r*}
  *E* = {Fact→*fact*, N→*n*, R→*r* }

- Here is what we will execute:

  {Fact N R}, E, σ

128

# Execution of {Fact 0} (2)

- To execute {Fact N R} we replace it by the procedure body and we replace the calling environment by a new environment
- The instruction:

{Fact N R}, {Fact→*fact*, N→*n*, R→*r* }, σ

(N,R: calling arguments)

is replaced by the instruction:

Later on we will see how to replace the calling environment by a new environment inside the procedure body.

**local** B **in**
B=(N==0)
**if** B **then** R=1 **else** … **end**
**end**, {Fact→*fact*, N→*n*, R→*r* }, σ

(N,R: arguments of Fact)

**Can be different from the calling environment!**

129

# Execution of {Fact 0} (3)

- To execute the **local** instruction:

**local** B **in**
B=(N==0)
**if** B **then** R=1 **else** … **end**
**end**, {Fact→*fact*, N→*n*, R→*r* }, σ

we do two operations:

- We extend the memory with a new variable *b*
- We extend the environment with {B → *b*}

- We then replace the instruction by its body:

B=(N==0)
**if** B **then** R=1 **else** … **end**,
{Fact→*fact*, N→*n*, R→*r*, B → *b*}, σ∪{*b*}

130

# Execution of {Fact 0} (4)

- We now do the same for:
  B=(N==0)
  and:
  **if** B **then** R=1 **else** … **end end**
- This will first bind *b*=**true** and then bind *r=1*
- This completes the execution of {Fact 0}

- We have executed {Fact 0} with the semantics and shown that the result is 1
- To complete the proof, we still have to show that the result of {Fact N} is the same as N*{Fact N-1}

131

# We have proved the correctness of Fact

- Let's recapitulate the approach
- Start with the specification and program of Fact
  - We want to prove that the program satisfies the specification
  - Since the function is recursive, our proof uses mathematical induction
- We need to prove the base case and the general case:
  - Prove that {Fact 0} execution gives 1
  - Prove that {Fact N} execution gives n × (result of {Fact N-1} execution)
- We prove both cases using the semantics and the program
  - To use the semantics, we first translate Fact into kernel language, and then we execute on the abstract machine
- This completes the proof

132

# Semantics 5: Semantic rules of procedures

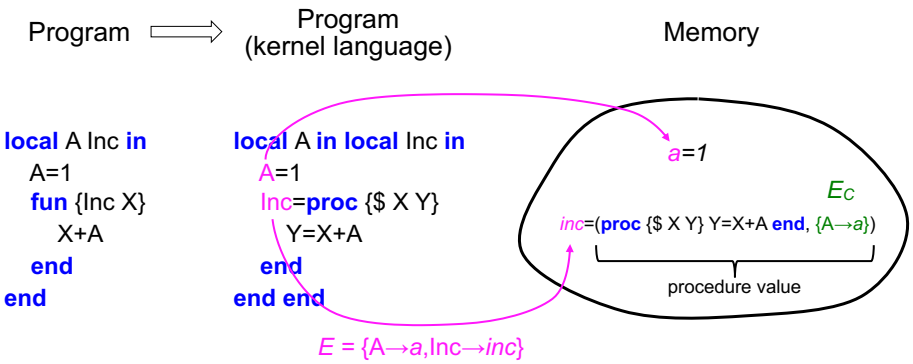
133

## Procedures are the building blocks of abstraction

- Procedure definition and call are very important, since they are the foundation of all data abstraction
  - Higher-order programming
  - Layered program organization
  - Encapsulation
  - Object-oriented programming
  - Abstract data types
  - Component-oriented programming
- This is why we study them separately

134

# We recall how procedures are stored in memory




135

# Defining and calling procedures

- Defining a procedure
  - Create the contextual environment
  - Store the procedure value, which contains both procedure code and contextual environment
- Calling a procedure
  - Create a new environment by combining two parts:
    - The procedure's contextual environment
    - The formal arguments (identifiers in the procedure definition), which are made to reference the actual argument values (at the call)
  - Execute the procedure body with this new environment
- We first give an example execution to show what the semantic rules have to do

136

# Procedure call example (1)

```
local Z in
   Z=1
   proc {P X Y} Y=X+Z end
end
```

- The free identifiers of the procedure (here, just Z) are the identifiers declared outside the procedure
- When executing P, the identifier Z must be known
- Z is part of the procedure's contextual environment, which must be part of the procedure value

137

# Procedure call example (2)

```
local P in
   local Z in
      Z=1
      proc {P X Y} Y=X+Z end  % E_C = {Z→z}
   end
   local B A in
      A=10
      {P A B}        % P's body Y=X+Z must do b=a+z
      {Browse B}     % Therefore: E_P = {Y→b, X→a, Z→z}
   end
end
```

138

# Semantic rule for procedure definition

- Semantic instruction:
  (<x>=**proc** {$ <x>$_1$ … <x>$_n$} <s> **end**, $E$)
  - Formal arguments:
    <x>$_1$, …, <x>$_n$
  - Free identifiers in <s>:
    <z>$_1$, …, <z>$_k$
  - Contextual environment:
    $E_C = E_{|\text{<z>1, ..., <z>k}}$ (restriction of $E$ to free identifiers)
- Create the following binding in memory:
  $x$=(**proc** {$ <x>$_1$ … <x>$_n$} <s> **end**, $E_C$)

139

# Semantic rule for procedure call (1)

- Semantic instruction:
  ({⟨x⟩ ⟨y⟩$_1$ … ⟨y⟩$_n$}, $E$)
- If the activation condition is false ($E$(⟨x⟩) unbound)
  - Suspension (do not execute, wait until $E$(⟨x⟩) is bound)
- If $E$(⟨x⟩) is not a procedure
  - Raise an error condition
- If $E$(⟨x⟩) is a procedure with the wrong number of arguments ($\neq n$)
  - Raise an error condition

140

# Semantic rule for procedure call (2)

Most important slide of the course

- Semantic instruction on stack:
  $(\{\langle x\rangle\ \langle y\rangle_1\ \ldots\ \langle y\rangle_n\}, E)$
  with procedure definition in memory:
  $E(\langle x\rangle) = (\textbf{proc}\ \{\$\ \langle z\rangle_1 \ldots \langle z\rangle_n\}\ \langle s\rangle\ \textbf{end}, E_C)$

- Put the following instruction on the stack:

  $(\langle s\rangle, E_C + \{\langle z\rangle_1 \rightarrow E(\langle y\rangle_1), \ldots, \langle z\rangle_n \rightarrow E(\langle y\rangle_n)\})$

141

# Computing with environments

- The abstract machine does two kinds of computations with environments
- Adjunction: $E_2 = E_1 + \{x \rightarrow y\}$
  - Add a pair (identifier→variable) to an environment
  - Overrides the same identifier in $E_1$ (if it exists)
  - Needed for **local** <x> **in** <s> **end** (and others)
- Restriction: $E_C = E_{|\{X,Y,Z\}}$
  - Limit identifiers in an environment to a given set
  - Needed to calculate the contextual environment

142

## Adjunction

- For a **local** instruction

```
local X in (E1)
    X=1
    local X in (E2)
        X=2
        {Browse X}
    end
end
```

- $E_1$ = {Browse → *b*, X → *x*}
- $E_2$ = $E_1$ + {X → *y*} = {Browse → *b*, X → *y*}

143

## Restriction

- For a procedure declaration

```
local A B C AddB in
    A=1 B=2 C=3 (E)
    fun {AddB X} (EC: contextual environment)
        X+B
    end
end
```

- $E$ = {A → *a*, B → *b*, C → *c*, AddB → *a'* }
- $E_C$ = $E_{|\{B\}}$ = {B → *b*}

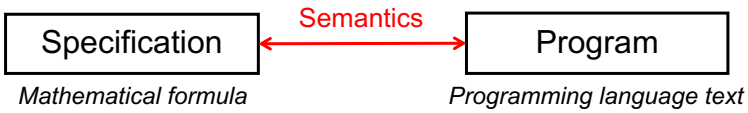
144

# Semantics summary

145

# Bringing it all together

- Defining the semantics brings many concepts together
  - Concepts we have seen before: identifier, variable, environment, memory, instruction, kernel language
  - New concepts: procedure value, semantic instruction, semantic stack, semantic rule, execution state, execution, abstract machine
- We gave semantic rules for the kernel language instructions, to show how they execute in the abstract machine
- We used the semantics to prove program correctness, by using it as bridge between specification and program




146

# Discrete mathematics

- The abstract machine is built with discrete mathematics
- It is probably the most complex construction that you have seen built with discrete mathematics!
  - Engineering students are quite used to integrals, differential equations, and complex analysis, which are all continuous mathematics, and the abstract machine is a new construction!
- Discrete mathematics is important because that's how computing systems work (both software and hardware)
  - Surprising behavior and bugs become less surprising if you understand the discrete mathematics of computing systems
  - Too often, continuous models are used for computing systems
  - All this applies to the real world as well (beyond computing systems)

147

# Why semantics is important

- Semantics is an intrinsic part of programming
  - As a programmer, you are extending the system's semantics: you are writing specifications, designing and implementing abstractions (which we will see later on), and reasoning about your work
- The design of any complicated system with parts that interact in interesting ways (like programming languages and programs) should be done hand in hand with designing a semantics
  - Designing a *simple* semantics is the only way to avoid unpleasant surprises and to guarantee a simple mental model
  - Users don't need to understand the semantics to take advantage of it: its mere existence is enough
    - Only the system's designers need to understand the semantics
- « Semantics is the ultimate programming language »
  - Invariants as the ultimate loop construct
  - Data abstractions as new kernel language instructions

148

## Using the semantics

- Semantics has many uses:
  - For design (ensuring the design is simple and predictable)
  - For understanding (the nooks and crannies of programs)
  - For verification (correctness and termination)
  - For debugging (a bug is only a bug with respect to a correct execution)
  - For visualization (a visual representation must be correct)
  - For education (pedagogical uses of semantics)
  - For program analysis and compiler design
- We don't need to bring in details of the processor architecture or compiler in order to understand many things about programs
  - For example, our semantics can be used to understand garbage collection
  - We will use the semantics when needed in the rest of the course

149

## "Semantics is the ultimate programming language"

150

# LINFO1104
# Concepts, paradigms, and semantics of programming languages

## Lecture 4

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

## Overview of lecture 3

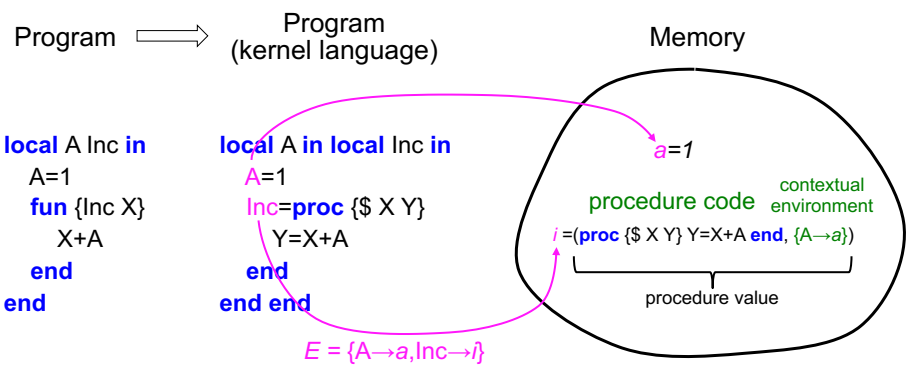
- Refresher of procedure semantics
- Higher-order programming
  - Order of a function
  - Genericity
  - Instantiation
  - Function composition
  - Abstracting an accumulator
  - Encapsulation
  - Delayed execution

2

# Procedure semantics refresher

3

# Procedure value in memory




4

# Procedure call

- Practical language:

  {Browse {Inc 10}}

- Kernel language:

```
local M in
   local N in
      M=10
      {Inc M N}
      {Browse N}
   end
end
```

5

5

# Execution of {Inc M N}

Important slide

- [({Inc M N},{M→*m*,N→*n*,Inc→*i*,Browse→*b*}),
  ({Browse N},{M→*m*,N→*n*,Inc→*i*,Browse→*b*})],
  σ = {*m*=10,*n*,*i*=(**proc** {$ X Y} Y=X+A **end**, {A→*a*}),
  *a*=1,*b*=(…browser code…)}

one execution step

Procedure body Y=X+A
New environment {A→*a*,X→*m*,Y→*n*} =
contextual environment {A→*a*}
+ formal arguments {X→*m*,Y→*n*}

- [(Y=X+A,{A→*a*,X→*m*,Y→*n*}),
  ({Browse N},{M→*m*,N→*n*,Inc→*i*,Browse→*b*})],
  σ

6

6

# Higher-order programming

7

# Higher-order programming

- Defining a procedure as a procedure value with a contextual environment is enormously expressive
  - It is arguably the most important invention in programming languages: it makes possible building large systems based on data abstraction
- Since procedures (and functions) are values, we can pass them as inputs to other functions and return them as outputs
  - Remember that in our kernel language, we consider functions and procedures to be the same concept: a function is a procedure with an extra output argument

8

## Order of a function

- We define the order of a function (or procedure)
  - A function whose inputs and output are not functions is first order
  - A function is order N+1 if its inputs and output contain a function of maximum order N
- Let's give some examples to show what we can do with higher-order functions (where the order is greater than 1)
  - We will give more examples later in the course

9

## Genericity

- Genericity is when a function is passed as an input

```
declare
fun {Map F L}
   case L of nil then nil
   [] H|T then {F H}|{Map F T}
   end
end

{Browse {Map fun {$ X} X*X end [7 8 9]}}
```

What is the order of Map in this call?

10

# Instantiation

- Instantiation is when a function is returned as an output

```
declare
fun {MakeAdd A}
    fun {$ X} X+A end
end
Add5={MakeAdd 5}

{Browse {Add5 100}}
```

What is the order of MakeAdd?

What is the contextual environment of the function returned by MakeAdd?

11

# Function composition

- We take two functions as input and return their composition

What is the contextual environment of the function returned by Compose?

```
declare
fun {Compose F G}
    fun {$ X} {F {G X}} end
end
Fnew={Compose fun {$ X} X*X end
              fun {$ X} X+1 end}
```

- What does {Fnew 2} return?
- What does {{Compose Fnew Fnew} 2} return?

12

# Abstracting an accumulator

- We can use higher-order programming to do a computation that hides an accumulator
- Let's say we want to sum the elements of a list $L=[a_0\ a_1\ a_2\ \ldots\ a_{n-1}]$:
  - $S = a_0 + a_1 + a_2 + \ldots + a_{n-1}$
  - $S = (\ldots(((0 + a_0) + a_1) + a_2) + \ldots + a_{n-1})$
- We can write this generically with a function F:
  - $S = \{F\ \ldots\ \{F\ \{F\ \{F\ 0\ a_0\}\ a_1\}\ a_2\}\ \ldots\ a_{n-1}\}$
- Now we can define the higher-order function FoldL:
  - $S = \{\text{FoldL}\ [a_0\ a_1\ a_2\ \ldots\ a_{n-1}]\ F\ 0\}$
  - The accumulator is hidden inside FoldL!

13

# Definition of FoldL

- Here is the definition of FoldL:

```
declare
fun {FoldL L F U}
   case L
   of nil then U
   [] H|T then {FoldL T F {F U H}}
   end
end
S={FoldL [5 6 7] fun {$ X Y} X+Y end 0}
```

The argument U is an accumulator

What is the order of FoldL?

14

# Encapsulation

- We can hide a value inside a function:

```
declare
fun {Zero} 0 end
fun {Inc H}
N={H}+1 in
      fun {$} N end
end
Three={Inc {Inc {Inc Zero}}}
{Browse {Three}}
```

- This is the foundation of encapsulation as used in data abstraction
- What is the difference if we write Inc as follows:

```
      fun {Inc H} fun {$} {H}+1 end end
```

15

# Delayed execution

- We can define an statement and pass it to a function which decides whether or not to execute it

```
proc {IfTrue Cond Stmt}
      if {Cond} then {Stmt} end
end
Stmt = proc {$} {Browse 111*111} end
{IfTrue fun {$} 1<2 end Stmt}
```

- This can be used to build control structures from scratch (**if** statement, **while** loop, **for** loop, etc.)

16

## Building a while loop (1)

- We build a generic while loop of this form:

```
s=init; while (cond(s)) s=transform(s);
```

All while loops can be written in this form

```
fun {While S Cond Transform}
   if {Cond S} then
      {While {Transform S} Cond Transform}
   else S end
end
```

17

## Building a while loop (2)

- Here is a while loop that sums integers from 1 to *n*
  - State is a pair s(I A) where I is the index and A is an accumulator
- {Browse {While s(10 0)
  **fun** {$ S} S.1>0 **end**
  **fun** {$ S} s(S.1-1 S.1+S.2) **end**}}
- Practical languages will define syntactic sugar for this:

```
i=10; a=0; while (i>0) { a=a+i; i--; }
```

18

# Summary of higher-order

- We have given six examples to illustrate the expressiveness of higher-order programming:
  - Genericity
  - Instantiation
  - Function composition
  - Abstracting an accumulator
  - Encapsulation
  - Delayed execution
- We will use these techniques and others when we introduce the concepts of data abstraction
  - Data abstraction is built on top of higher-order programming!

19

# LINFO1104
# Concepts, paradigms, and semantics of programming languages

## Lecture 5

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

## Overview of lecture 5

- Refresher of higher-order programming
  - Higher-order programming is a key technique
  - Many powerful techniques are possible, and we will use them to build abstractions
- Lambda calculus
  - A very simple model of computation that is Turing equivalent
  - All data types can be encoded in lambda calculus
  - Church-Rosser theorem: lambda calculus is confluent, i.e., same results for all reduction orders
    - The key reason why functional programming is an important paradigm
  - The foundation of higher-order programming and functional programming languages

2

# Introduction

- Lambda calculus is a formal system in mathematical logic for expressing computation
  - It is based on function abstraction and application, using variable binding and substitution
  - It was introduced by logician Alonzo Church in the 1930s as part of research into the foundations of mathematics
- Lambda calculus is a universal model of computation that can be used to simulate a Turing machine
  - Untyped lambda calculus, introduced by Church in 1936
  - In the 1960s, the relation to programming languages was clarified (Peter Landin 1965) and it has since been used as a foundation for understanding and designing programming languages

3

# Basic concepts

- The lambda calculus provides a simple semantics for computation
- The lambda calculus has only anonymous functions of one argument:
  sum_square(x,y) = $x^2+y^2$
  becomes an anonymous function:
  $(x,y) \rightarrow x^2+y^2$
  of one argument:
  $x \rightarrow (y \rightarrow x^2+y^2)$
- Converting functions into nested one-argument functions is called currying

4

# Syntax of lambda expressions

- Lambda expressions are composed of:
  - Variables *x*, *y*, …
  - Abstraction symbols λ (lambda) and . (dot)
  - Parentheses
- Lambda terms *t* are defined with the following rule:

  $t ::= x \mid (\lambda x.\ t) \mid (t_1\ t_2)$

- Terminology:
  - $(\lambda x.\ t)$ is called an *abstraction*
  - $(t_1\ t_2)$ is called an *application*

5

# Lambda expressions in Oz

- Lambda expressions in Oz:
  - λ*x*. *t*
    **fun** {$ X} *T* **end**
  - $(t_1\ t_2)$
    $\{T_1\ T_2\}$
- Currying in Oz:
    F=**fun** {$ X Y} T **end**
  becomes:
    F=**fun** {$ X} **fun** {$ Y} T **end end**
- The call {F X Y} becomes {{F X} Y}

6

# Free and bound variables

- The abstraction operator λ binds its variable when it occurs in the body of the abstraction
- In a term $\lambda x.t$, the part $\lambda x$ is called the *binder*, and it *binds* the variable $x$ in $t$
- The set of free variables in a term $t$ is denoted as FV($t$) and is defined as follows:
  FV($x$)={$x$} where $x$ is a variable
  FV($\lambda x.t$)=FV($t$)\{$x$}
  FV(($t_1$ $t_2$))=FV($t_1$) ∪ FV($t_2$)

7

# Semantics of lambda expressions

- The meaning of lambda terms is defined by how they can be reduced
- There are three possible reductions:
  - α-renaming: change bound variable names
  - β-reduction: apply functions to arguments
  - η-reduction: remove unused variables (extensionality)
- We show reduction steps with arrows:
  $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_{n-2} \rightarrow t_{n-1} \rightarrow \cdots$
  - We write $t_i \rightarrow^* t_j$ for zero or more reductions
  - We write $t_i \rightarrow_\beta t_j$ for a β-reduction

8

## α-renaming (α-conversion)

- Allows bound variable names to be changed:
  - Example: $\lambda x.x \rightarrow \lambda y.y$
- A name can be changed only if it does not introduce a name conflict (in λx.t, the relationship between λx and the bound variables in t must be unchanged):
  - It cannot change a bound variable in a subterm
    - Allowed λx.λx.x → λy.λx.x but not allowed λx.λx.x ↛ λy.λx.y
  - It cannot do capturing
    - Not allowed λx.λy.x ↛ λy.λy.y because y is captured
- Terms differing by α-renaming are called α-equivalent
  - Terms that are α-equivalent form an equivalence class

9

## Substitution

- Substitution $t_1[x:=t_2]$ replaces all free occurrences of x in $t_1$ by $t_2$
- Definition:
  - $x[x:=t] = t$
  - $y[x:=t] = y$, if $x \neq y$
  - $(t_1\ t_2)[x:=t] = (t_1[x:=t])\ (t_2[x:=t])$
  - $(\lambda x.t_1)[x:=t_2] = \lambda x.t_1$
  - $(\lambda y.t_1)[x:=t_2] = \lambda y.(t_1[x:=t_2])$, if $x \neq y$ and $y \notin FV(t_2)$
- α-renaming is allowed to make substitution possible
  - Substitution defined up to α-equivalence

10

## β-reduction

- β-reduction is the idea of function application
  - It is defined in terms of substitution
- Definition:
  $(\lambda x.t_1)\ t_2 \ \rightarrow\ t_2[x:=t_1]$
- Example: $(\lambda x.(x\ x))\ y \rightarrow (y\ y)$

11

## η-reduction

- η-reduction expresses the idea that two functions are the same if and only if they have the same results for all arguments
- Definition:
  $\lambda x.(t\ x) \rightarrow t$ if $x \notin FV(t)$
- Example: $(\lambda x.(t\ x))\ a \rightarrow (t\ a)$ *(if x is not free in t)*
  It means that $\lambda x.(t\ x)$ is the same function as t

12

## Summary of reduction rules

- α-renaming
  $\lambda x.t_1[x] \rightarrow \lambda y.t_2[y]$
  (change bound vars without capture)
- β-reduction
  $(\lambda x.t_1)\ t_2 \rightarrow t_2[x:=t_1]$
- η-reduction
  $\lambda x.(t\ x) \rightarrow t$ if $x \notin FV(t)$
- Examples on the board!

13

## Notation conventions

- Important when writing and manipulating lambda expressions!
  - Drop outermost parentheses: $(t_1\ t_2) \rightarrow t_1\ t_2$
  - Applications are left-associative: $t_1\ t_2\ t_3 \rightarrow (t_1\ t_2)\ t_3$
  - Abstraction body extends right: $\lambda x.t_1\ t_2$ means $\lambda x.(t_1\ t_2)$
  - Sequence of abstractions: $\lambda x.\lambda y.\lambda z.t$ written as $\lambda xyz.t$
- You will see why this is important when we start manipulating big lambda expressions
  - We will also use abbreviations a lot, it really helps when doing lambda computations by hand

14

# Encoding datatypes

- The untyped lambda calculus can be used to do all computations
  - It is actually Turing equivalent
- One way to show this is to encode datatypes and control operations as lambda terms
  - Numbers and arithmetic in lambda calculus
  - Boolean operations and conditional (if statement) in lambda calculus
  - Lists in lambda calculus (data structures)
  - Recursive functions in lambda calculus

15

# Arithmetic: numbers

- Encoding natural numbers (Church numerals):
  - $0 \triangleq \lambda f.(\lambda x.x)$
  - $1 \triangleq \lambda f.(\lambda x.(f\ x))$
  - $2 \triangleq \lambda f.(\lambda x.(f\ (f\ x)))$
  - $3 \triangleq \lambda f.(\lambda x.(f\ (f\ (f\ x))))$
  - (we use the symbols 0, 1, 2, 3 as short-cuts for the lambda terms) (symbol "$\triangleq$" means "is defined as")
- A Church numeral is a higher-order function: it takes a single-argument function f and returns another single-argument function
- The Church numeral n is a function that takes a function f as argument and returns the n-th composition of f, i.e., f composed with itself n times: it is like saying "f is applied n times"

16

## Arithmetic: operations

- Successor function takes Church numeral n and returns n+1:
  SUCC ≜ λn.λf.λx.f ((n f) x)
  SUCC ≜ λn.λf.λx.f (n f x)
- Addition (plus) uses the fact that the m-th composition of f composed with the n-th composition of f gives the (m+n)-th composition of f:
  PLUS ≜ λm.λn.λf.λx.(m f) ((n f) x)
  PLUS ≜ λm.λn.λf.λx.m f (n f x)

17

## Example of successor

- SUCC 1 ≡
  (λn.λf.λx.f (n f x)) λf.(λx.(f x) →
  λf.λx.f ((λf.(λx.(f x)) f x) →
  λf.λx.f (f x) ≡
  2
- We have incremented 1!
- The symbol "≡" means "is equivalent to"
  - We can always replace an abbreviation by the lambda expression it is equivalent to

18

# Arithmetic: operations

- We can verify that PLUS 2 3 is equivalent to 5
- Multiplication can be defined as:
  MULT ≜ λm.λn.λf.m (n f)
  or else:
  MULT ≜ λm.λn.m (PLUS n) 0
  "repeat m times the 'PLUS n' starting with 0"
- Exponentiation can be defined as:
  POW ≜ λb.λe.e b

19

# Arithmetic: operations

- The predecessor function, defined as n-1 for a positive integer n, and PRED 0 = 0, is much harder:

  PRED ≜ λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)

- With PRED we can define subtraction:
  SUB ≜ λm.λn.n PRED m
- The term SUB m n yields m-n when m>n and 0 otherwise

20

# Logical operations

- By convention, we express true and false as follows:
  TRUE ≜ λx.λy.x
  FALSE ≜ λx.λy.y
- We can define logic operators:
  AND ≜ λp.λq.p q p
  OR ≜ λp.λq.p p q
  NOT ≜ λp.p FALSE TRUE
  IFTHENELSE ≜ λp.λa.λb.p a b
- For example: (example on the board)
  AND TRUE FALSE
  ≡ (λp.λq.p q p) TRUE FALSE $\rightarrow_\beta$ TRUE FALSE TRUE
  ≡ (λx.λy.x) FALSE TRUE $\rightarrow_\beta$ FALSE

21

# Predicates

- A predicate is a function that returns a boolean value
- Compare with zero:
  Return TRUE if the argument is 0 and FALSE if the argument is any other number:
  ISZERO ≜ λn.n (λx.FALSE) TRUE
- Less-than-or-equal:
  LEQ ≜ λm.λn.ISZERO (SUB m n)

22

## Pairs (cons cells)

- A pair is a 2-tuple, it can be defined in terms of TRUE and FALSE
  - PAIR encapsulates the pair (x,y), FIRST returns the first element, and SECOND returns the 2nd
- PAIR ≜ λx.λy.λf.f x y
  FIRST ≜ λp.p TRUE
  SECOND ≜ λp.p FALSE
  NIL ≜ λx.TRUE
  NULL ≜ λp.p (λx.λy.FALSE) (test if nil, return TRUE or FALSE)

23

## Using pairs

- A list is either NIL (empty list) or the PAIR of an element and a smaller list
- Example of use of pairs:
  SHIFTINC ≜ λx.PAIR (SECOND x) (SUCC (SECOND x))
  - Maps (m,n) to (n,n+1)
- With SHIFTINC we can define predecessor in a simpler way:
  PRED ≜ λn.FIRST (n SHIFTINC (PAIR 0 0))

24

## Recursive functions (1)

- Since functions are anonymous, we can't do recursion directly in lambda calculus
  - However, we can do recursion by arranging for a lambda term to get itself as its argument
  - Kind of like this in Oz:

```
Fact=fun {$ F N}
        if N==0 then 1
        else N*{F F N-1} end
     end
{Browse {Fact Fact 5}} % Displays 120
```

25

## Recursive functions (2)

- We show how to do recursive factorial in lambda calculus
  - We start with a factorial written using the data and control structures we already defined:
    Fact(n) := if n=0 then 1 else n × Fact(n-1)
- We rewrite this so it becomes a function of two arguments, where the first argument is the function itself:
  G := λr. λn. (if n=0 then 1 else n × (r n-1))

26

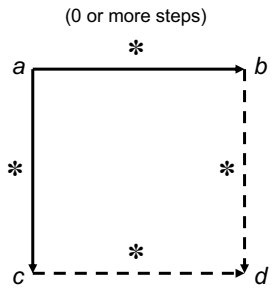
# Recursive functions (3)

- We define a helper called "Y combinator":
  $Y \triangleq \lambda g.(\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))$
  - We can show: $Y\ g \rightarrow^* g\ (Y\ g)$
  - This extracts the g and calls it with argument (Y g)
    - Remember that g has two arguments, so g (Y g) is a one-arg function!
- Now we can do the recursive factorial:
  $(Y\ G)\ 4 \rightarrow^* G\ (Y\ G)\ 4 \rightarrow$
  $(\lambda r.\ \lambda n.(\text{if } n=0 \text{ then } 1 \text{ else } n \times (r\ n\text{-}1))\ (Y\ G)\ 4 \rightarrow$
  $\text{if } 4=0 \text{ then } 1 \text{ else } 4 \times ((Y\ G)\ 4\text{-}1) \rightarrow$
  $4 \times (G\ (Y\ G)\ 4\text{-}1) \rightarrow \ldots \rightarrow$
  $4 \times 3 \times 2 \times 1 \times 1 \rightarrow 24$

27

# Church-Rosser theorem

- An amazing property of lambda calculus is that the order of reduction makes no difference
- Church-Rosser theorem:
  If *a* reduces to *b* (in 0 or more steps), and *a* reduces to *c* (in 0 or more steps), then there exists a term *d* such that *b* and *c* can reduce to *d*
- We say that the lambda calculus is *confluent* or that it has the *Church-Rosser property*
- It means that the result of a computation is the same no matter in what order the reductions are done




28

# Lambda calculus and programming languages

- Functional programming languages can be understood in terms of the lambda calculus
  - Procedure values (lexically scoped closures) are lambda functions
  - Eager versus lazy evaluation is just a different reduction strategy
- Reduction strategies define in what order expressions are computed:
  - Applicative order: leftmost innermost first (arguments evaluated before function)
    - Call by value (eager evaluation): similar to applicative order, but no reductions inside function definitions (compiled functions are not changed)

Traditional langs. (Java, Python)

  - Normal order: leftmost outermost first (function evaluated before arguments)
    - Call by name: similar to normal order except no reductions inside function definitions (compiled functions are not changed)
    - Call by need (lazy evaluation): similar to normal order except that functions are shared, not copied (evaluated at most once)

Lazy functional langs (Haskell)

29

# Applicative order example

- **fun** {Double X} X+X **end**
  **fun** {Average X Y} (X+Y)/2 **end**

  {Double {Average 5 7}} →
  {Double ((5+7)/2)} →
  {Double (12/2)} →
  {Double 6} →
  6+6 →
  12

Many popular languages use applicative order (Java, Python, C++, etc.)

30

# Normal order example

- **fun** {Double X} X+X **end**
  **fun** {Average X Y} (X+Y)/2 **end**

  {Double {Average 5 7}} →
  {Average 5 7}+{Average 5 7} →
  ((5+7)/2)+{Average 5 7} →
  (12/2)+{Average 5 7} →
  6+{Average 5 7} →
  6+((5+7)/2) →
  6+(12/2) →
  6+6 →
  12

In contrast to applicative order, normal order only evaluates a function if its result is needed for the computation

31

# Lazy evaluation example

- **fun** {Double X} X+X **end**
  **fun** {Average X Y} (X+Y)/2 **end**

  {Double {Average 5 7}} →
  **local** X={Average 5 7} **in** X+X **end** →
  **local** X=((5+7)/2) **in** X+X **end** →
  **local** X=(12/2) **in** X+X **end** →
  **local** X=6 **in** X+X **end** →
  6+6 →
  12

Lazy evaluation is similar to normal order except that the Average function is only evaluated once (the two copies are shared)

The functional language Haskell uses lazy evaluation by default; some functional languages use eager evaluation by default but allow declaring lazy evaluation (OCaml, Oz)

32

# If statement evaluation

- **if** 3<4 **then** 5+5 **else** 1/0 **end**

- Normal order:

  **if** 3<4 **then** 5+5 **else** 1/0 **end** →
  **if true then** 5+5 **else** 1/0 **end** →
  5+5 →
  10

- Applicative order:

  **if** 3<4 **then** 5+5 **else** 1/0 **end** →
  **if true then** 5+5 **else** 1/0 **end** →
  **if true then** 10 **else** 1/0 **end** →
  [Error: Division by zero]

- Most languages do **if** statements with applicative order for the condition and with normal order for the **then** and **else** parts!

33

# Variations and extensions

- Lambda calculus is a fundamental part of theoretical computer science research
- Theoretical work on programming languages defines many extensions of the lambda calculus:
  - Typed lambda calculus: lambda calculus with typed variables and functions
  - System F: typed lambda calculus with type variables (variables ranging over types)
  - Calculus of constructions: typed lambda calculus with types as first-class values
  - Combinatory logic: logic without variables
  - SKI combinator calculus: equivalent to lambda calculus but without variable substitutions
  - Oz kernel language: a lambda calculus with futures (single-assignment variables), dataflow synchronization, threads, and explicit lazy evaluation

34

# Summary

- The lambda calculus is part of the theoretical foundation of almost all programming languages
  - All except for logic and constraint languages, which are based on formal logic
  - Lambda calculus was introduced by Alonzo Church (1936)
  - Its relationship to programming was first recognized by Peter Landin (1965)
    - It is the reason why languages from the 1950s (Fortran and Lisp) did not do functions right!
- The lambda calculus has strong properties
  - Lambda calculus is Turing equivalent (all computations can be expressed)
  - Church-Rosser theorem: The result of a computation is independent of the reduction strategy (this is also known as confluence)
    - Important reduction strategies are eager evaluation, lazy evaluation, and dataflow concurrency
  - Higher-order programming is a consequence of the lambda calculus
    - It is the foundation of language abstraction (objects, classes, ADTs, components, etc.)
  - Functional programming languages (Haskell, Scheme, Scala, OCaml, Oz, etc.) are designed to take advantage of these properties

# LINFO1104
# Concepts, paradigms, and semantics of programming languages

## Lecture 6 & 7

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

## Overview of lectures 6 and 7

- Today is the midterm!
  - First hour of the course. Good luck!
- Mutable state
  - So far we have done pure functional programming
  - Functional programming is modeled by the lambda calculus and has strong mathematical properties
  - But it's not enough! The same thing that makes it powerful (functions cannot change) is a weakness (sometimes functions must change).
- Data abstraction
  - Data abstraction is the main organizing principle for building complex software systems
  - Data abstraction is built upon two concepts: higher-order programming and mutable state

2

# Time and change in programs

3

## Time and change

- In functional programming, there is no notion of time
  - All functions are mathematical functions; once defined they never change
  - Programs do execute on a real machine, but a program cannot observe the execution of another program or of part of itself
    - It can only see the results of a function call, not the execution itself
    - Observing an execution of a program can only be done outside of the program's implementation
- In the real world, there is time and change
  - Organisms change their behavior over time, they grow and learn
  - How can we model this in a program?
- We need to add time to a program
  - Time is a complicated concept! Let us start with a simplified version of time, an abstract time, that keeps the essential property that we need: modeling change.

4

# State as an abstract time (1)

- Here's one solution: We define the abstract time as a sequence of values and we call it a state
- A state is a sequence of values calculated progressively, which contains the intermediate results of a computation
- The functional paradigm can use state according to this definition!
- The definition of Sum given here has a state

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

{Browse {Sum [1 2 3 4] 0}}
```

5

# State as an abstract time (2)

- The two arguments Xs and A give us an implicit state

| Xs | A |
|---|---|
| [1 2 3 4] | 0 |
| [2 3 4] | 1 |
| [3 4] | 3 |
| [4] | 6 |
| nil | 10 |

- It is implicit because the language has not changed
  - It is purely in the programmer's head: the programmer observes the changes in the program
- In most cases this is not good enough: we want the program itself to observe the changes
  - We need a language extension!
  - *We leave the functional paradigm and enter another paradigm*

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

{Browse {Sum [1 2 3 4] 0}}
```

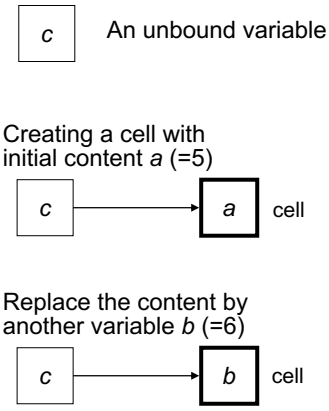
6

# Explicit state (mutable state)

7

# Adding mutable state to the language

- We can make the state explicit by extending the language
- With this extension a program can directly observe the sequence of values in time
  - This was not possible in the functional paradigm
- We call our extension a cell
  - The word "cell" is chosen to avoid confusion with related terms, such as the overused word "variable"
- A cell is a box with a content
  - The content can be changed but the box remains the same
  - The same cell can have different contents: we can observe change
  - The sequence of contents is a state

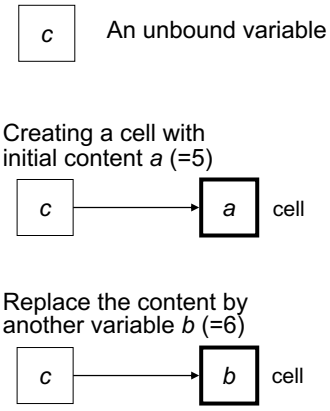



8

# A cell

- A cell is a box with an identity and a content
  - The identity is a constant
    (the “name” or “address” of the cell)
  - The content is a variable
    (in the single-assignment store)
- The content can be replaced by another variable

```
A=5
B=6
C={NewCell A}  % Create a cell
{Browse @C}    % Display content
C:=B           % Change content
{Browse @C}    % Display content
```




9

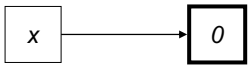
# Adding cells to the kernel language

- We add cells and their operations
  - Cells have three operations
- C={NewCell A}
  - Create a new cell with initial content A
  - Bind C to the cell’s identity
- C:=B
  - Check that C is bound to a cell’s identity
  - Replace the cell’s content by B
- Z=@C
  - Check that C is bound to a cell’s identity
  - Bind Z to the cell’s content

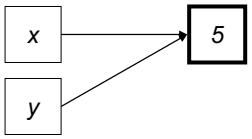
10

## Some examples (1)

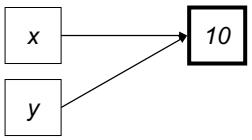
- X={NewCell 0}




- X:=5
- Y=X



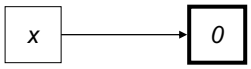

- Y:=10
- @X==10 % true
- X==Y % true

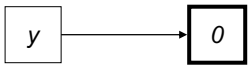



11

## Some examples (2)

- X={NewCell 0}
- Y={NewCell 0}




- X==Y % false
- Because X and Y refer to different cells, with different identities

- @X==@Y % true
- Because the contents of X and Y are the same value

12

# Semantics of cells

13

# Semantics of cells (1)

- We have extended the kernel language with cells
  - Let us now extend the abstract machine to explain how cells execute
- There are now two stores in the abstract machine:
  - Single-assignment store (contains variables: immutable store)
  - Multiple-assignment store (contains cells: mutable store)
- A cell is a pair of two variables
  - The first variable is bound to the name of the cell (a constant)
  - The second variable is the cell's content
- Assigning a cell to a new content
  - The pair is changed: the second variable in the pair is replaced by another variable (the first variable stays the same)

Warning: The variables do *not* change! The single-assignment store is unchanged when a cell is assigned.

14

## Semantics of cells (2)

- The full store $\sigma = \sigma_1 \cup \sigma_2$ has two parts:
  - Single-assignment store (contains variables)
    $\sigma_1$ = {*t*, *u*, *v*, *x*=ξ, *y*=ζ, *z*=10, *w*=5}
  - Multiple-assignment store (contains pairs)
    $\sigma_2$ = {*x*:*t*, *y*:*w*}
- In $\sigma_2$ there are two cells, *x* and *y*
  - The name of *x* is the constant ξ, the name of *y* is ζ
  - The operation X:=Z changes *x*:*t* into *x*:*z*
  - The operation @Y returns the variable *w*
    (assuming the environment {X → *x*, Y → *y*, Z → *z*, W → *w*})

15

## Imperative programming

- By adding cells, we have left functional programming and entered imperative programming
  - Imperative paradigm = functional paradigm + cells
- Imperative programming allows programs to express and observe growth and change
  - This gives new ways of thinking that were not possible in functional programming
- Imperative programming is important for object-oriented programming (OOP)
  - OOP has new ways of structuring programs that are essential for building large systems

16

# Kernel language of imperative programming

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {NewCell <y> <x>}
  | <x>:=<y>
  | <y>=@<x>
- <v> ::= <number> | <procedure> | <record>
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <record>, <p> ::= <lit> | <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

17

# Kernel language of imperative programming

- <s> ::= **skip**
  | <s>$_1$ <s>$_2$
  | **local** <x> **in** <s> **end**
  | <x>$_1$=<x>$_2$
  | <x>=<v>
  | **if** <x> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {<x> <y>$_1$ … <y>$_n$}
  | **case** <x> **of** <p> **then** <s>$_1$ **else** <s>$_2$ **end**
  | {NewCell <y> <x>}
  | {Exchange <x> <y> <z>}
- <v> ::= <number> | <procedure> | <record>
- <number> ::= <int> | <float>
- <procedure> ::= **proc** {$ <x>$_1$ … <x>$_n$} <s> **end**
- <record>, <p> ::= <lit> | <lit>(<f>$_1$:<x>$_1$ … <f>$_n$:<x>$_n$)

| Second version |
| --- |
| Both versions are equally expressive (since Exchange can be expressed with @ and := and vice versa), but the second version is more convenient for concurrent programming |

<y>=@<x> and <x>:=<z> (*atomically,* i.e., as one operation)

18

# Mutable state is needed for modularity

19

## Mutable state is needed for modularity

- Before looking at data abstraction and object-oriented programming, let's take a closer look at what mutable state is good for
- We say that a program (or system) is modular with respect to a given part if that part can be changed without changing the rest of the program
  - "part" = function, procedure, component, module, class, library, package, file, …
- We will show by means of an example that the use of mutable state allows us to make a program modular
  - This is not possible in the functional paradigm

20

# A scenario (1)

- Once upon a time there were three developers, P, U1, and U2
- P has developed module M that implements two functions F and G
- U1 and U2 are both happy users of module M

```
fun {MF}  % Module definition
   fun {F ...}
      ⟨Definition of F⟩
   end
   fun {G ...}
      ⟨Definition of G⟩
   end
in 'export'(f:F g:G)
end
M = {MF} % Module instantiation
```

21

# A scenario (2)

- One day, developer U2 writes an application that runs slowly because it does too much computation
- U2 would like to extend M to count the number of times F is called by the application
- U2 asks P to make this extension, but to keep it modular so that no programs have to be changed to use it

```
fun {MF}
   fun {F ...}
      ⟨Definition of F⟩
   end
   fun {G ...}
      ⟨Definition of G⟩
   end
in 'export'(f:F  g:G)
end
M = {MF}
```

22

# Oops!

- This is impossible in functional programming, because F does not remember what happened in previous calls: it cannot count its calls
  - The only solution is to change the interface of F by adding two arguments, $F_{in}$ and $F_{out}$:
    **fun** {F … $F_{in}$ $F_{out}$} $F_{out}$=$F_{in}$+1 … **end**
  - The rest of the program has to make sure that the $F_{out}$ of each call to F is passed as $F_{in}$ to the next call of F
- This means that M's interface has changed
- All M's users, even U1, have to change programs
  - U1 is especially unhappy, since it makes a lot of extra work for nothing

23

# Solution using a cell

- Create a cell when MF is called and increment it inside F
  - Because of static scope, the cell is hidden from the rest of the program: it is only visible inside M
- M's interface is extended without changing existing calls
  - M.f stays the same
  - A new function M.c appears that can safely be ignored
- P, U1, and U2 live happily ever after

```
fun {MF}
   X = {NewCell 0}
   fun {F ...}
         X:=@X+1
         ⟨Definition of F⟩
   end
   fun {G ...}
         ⟨Definition of G⟩
   end
   fun {Count} @X end
in 'export'(f:F g:G c:Count)
end
M = {MF}
```

24

# Comparison

- Functional programming:
  - + A component never changes its behavior (correctness is permanent)
  - – Updating a component often means that its interface changes and therefore many other components must be updated
- Imperative programming:
  - + A component can be updated without changing its interface and so without changing the rest of the program (modularity)
  - – A component can change its behavior because of past calls (for example, it might break)
- Sometimes it is possible to combine both advantages
  - Use mutable state to manage updates, but make sure that the behavior of components does not change

25

# Data abstraction

26

## Data abstraction

- Data abstraction is the main organizing principle for building complex software systems
  - Without data abstraction, computing technology would stop dead in its tracks
- We will study what data abstraction is and how it is supported by the programming language
  - The first step toward data abstraction is called encapsulation
  - Data abstraction is supported by language concepts such as higher-order programming, static scoping, and explicit state

27

## Encapsulation

- The first step toward data abstraction, which is the basic organizing principle for large programs, is encapsulation
- Assume your television set is not enclosed in a box
  - All the interior circuitry is exposed to the outside
  - It's lighter and takes up less space, so it's good, right? NO!
- It's dangerous for you: if you touch the circuitry, you can get an electric shock
- It's bad for the television set: if you spill a cup of coffee inside it, you can provoke a short-circuit
  - If you like electronics, you may be tempted to tweak the insides, to "improve" the television's performance
- So it can be a good idea to put the television in an enclosing box
  - A box that protects the television against damage and that only authorizes proper interaction (on/off, channel selection, volume)

28

# Encapsulation in a program

- Assume your program uses a stack with the following implementation:

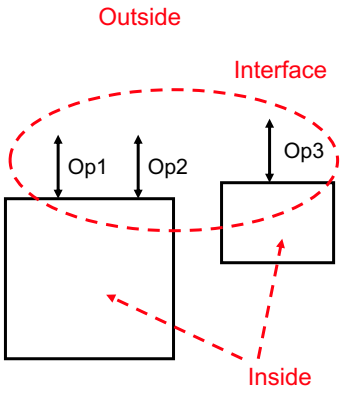
```
fun {NewStack} nil end
fun {Push S X} X|S end
fun {Pop S X} X=S.1 S.2 end
fun {IsEmpty S} S==nil end
```

- This implementation is not encapsulated!
  - It has the same problems as a television set without enclosure
  - It is implemented using lists that are not protected
    - A user can read stack values without the implementation knowing
    - A user can create stack values outside of the implementation
- There is no way to guarantee that an unencapsulated stack will work correctly
  - The stack must be encapsulated → data abstraction

29

# Definition of data abstraction

- A data abstraction is a part of a program that has an inside, an outside, and an interface in between
- The inside is hidden from the outside
  - All operations on the inside must pass through the interface, i.e., the data abstraction must use encapsulation
- The interface is a set of operations that can be used according to certain rules
  - Correct use of the rules guarantees that the results are correct
- The encapsulation must be supported by the programming language
  - We will see how the language can support encapsulation, that is, how it can enforce the separation between inside and outside




30

# Advantages of data abstraction

- A guarantee that the abstraction will work correctly
  - The interface only allows well-defined interaction with the inside
- A reduction of complexity
  - The user does not have to know the implementation, but only the interface, which is generally much simpler
  - A program can be partitioned into many independent abstractions, which greatly simplifies use
- The development of large programs becomes possible
  - Each abstraction has a responsible developer: the person who implements it, maintains it, and guarantees its behavior
  - Each responsible developer only has to know the interfaces of the abstractions used by the abstraction
  - It's possible for teams of developers to develop large programs

31

# The two main kinds of data abstraction

- There are two main kinds of data abstraction, namely objects and abstract data types
  - An object groups together value and operations in a single entity
  - An abstract data type keeps values and operations separate
- Some real world examples
  - A television set is an object: it can be used directly through its interface (on/off, channel selection, volume control)
  - Coin-operated vending machines are abstract data types: the coins and products are the values and the operations are the vending machines
- We will look at both objects and ADTs
  - Each has its own advantages and disadvantages

32

# Abstract data types

33

## Abstract data types

- An ADT consists of a set of values and a set of operations
- A common example: integers
  - Values: 1, 2, 3, …
  - Operations: +, -, *, div, …
- In most of the popular uses of ADTs, the values and operations have no state
  - The values are constants
  - The operations have no internal memory (they don't remember anything in between calls)

34

# A stack ADT

- We can implement a stack as an ADT:
  - Values: all possible stacks and elements
  - Operations: NewStack, Push, Pop, IsEmpty
- The operations take (zero or more) stacks and elements as input and return (zero or more) stacks and elements as output
  - S={NewStack}
  - S2={Push S X}
  - S2={Pop S X}
  - {IsEmpty S}
- For example:
  - S={Push {Push {NewStack} a} b} returns the stack S=[b a]
  - S2={Pop S X} returns the stack S2=[a] and the top X=b

35

# Unencapsulated implementation

- The stack we saw before is almost an ADT:
  - **fun** {NewStack} nil **end**
  - **fun** {Push S X} X|S **end**
  - **fun** {Pop S X} X=S.1 S.2 **end**
  - **fun** {IsEmpty S} S==nil **end**
- Here the stack is represented by a list
- But this is not a data abstraction, since the list is not protected
- How can we protect the list, and make this a true ADT?
  - How can we build an abstract data type with encapsulation?
  - We need a way to protect values

36

# Encapsulation using a secure wrapper

- To protect the values, we will use a secure wrapper:
  - The two functions Wrap and Unwrap will "wrap" and "unwrap" a value
  - W={Wrap X}  % Given X, returns a protected version W
  - X={Unwrap W}  % Given W, returns the original value X
- The simplest way to understand this is to consider that Wrap and Unwrap do encryption and decryption using a shared key that is only known by them
- We need a new Wrap/Unwrap pair for each ADT that we want to protect, so we use a procedure that creates them:
  - {NewWrapper Wrap Unwrap} creates the functions Wrap and Unwrap
  - Each call to NewWrapper creates a pair with a new shared key
- We will not explain here how to implement NewWrapper, but if you are curious you can look in the book (Section 3.7.5)

37

# Implementing the stack ADT

- Now we can implement a true stack ADT:

```
local Wrap Unwrap in
      {NewWrapper Wrap Unwrap}

      fun {NewStack} {Wrap nil} end
      fun {Push W X} {Wrap X|{Unwrap W}} end
      fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
      fun {IsEmpty W} {Unwrap W}==nil end
end
```

- How does this work?  Look at the Push function: it first calls {Unwrap W}, which returns a stack value S, then it builds X|S, and finally it calls {Wrap X|S} to return a protected result
- Wrap and Unwrap are hidden from the rest of the program (static scoping)

38

# Final remarks on ADTs

- ADT languages have a long history
  - The language CLU, developed by Barbara Liskov and her students in 1974, is the first
  - This is only a little bit later than the first object-oriented language Simula 67 in 1967
  - Both CLU and Simula 67 strongly influenced later object-oriented languages up to the present day
- ADT languages support a protection concept similar to Wrap/Unwrap
  - CLU has syntactic support that makes the creation of ADTs very easy
- Many object-oriented languages also support ADTs
  - For example, Java supports ADTs: Java integers are ADTs, and Java objects have some ADT properties

39

# Objects

40

# Objects

- A single object represents both a value and a set of operations
- Example interface of a stack object:

```
S={NewStack}
{S push(X)}
{S pop(X)}
{S isEmpty(B)}
```

- The stack value is stored inside the object S
- Example use of a stack object:

```
S={NewStack}
{S push(a)}
{S push(b)}
local X in {S pop(X)} {Browse X} end
```

41

# Implementing the stack object

- Implementation of the stack object:

```
fun {NewStack}
      C={NewCell nil}
      proc {Push X} C:=X|@C end
      proc {Pop X} S=@C in C:=S.2 X=S.1 end
      proc {IsEmpty B} B=(@C==nil) end
in
      proc {$ M}
         case M of push(X) then {Push X}
         [] pop(X) then {Pop X}
         [] isEmpty(B) then {IsEmpty B} end
      end
end
```

- Each call to NewStack creates a new stack object
- The object is represented by a one-argument procedure that does procedure dispatching: a case statement chooses the operation to execute
- Encapsulation is enforced by hiding the cell with static scoping

42

# Stack as ADT and stack as object

- Here is the stack as ADT:

```
local Wrap Unwrap in
        {NewWrapper Wrap Unwrap}
        fun {NewStack} {Wrap nil} end
        fun {Push W X} {Wrap X|{Unwrap W}} end
        fun {Pop W X} S={Unwrap W} in X=S.1 {Wrap S.2} end
        fun {IsEmpty W} {Unwrap W}==nil end
end
```

- Here is the stack as object: (represented by a record)

```
fun {NewStack}
        C={NewCell nil}
        proc {Push X} C:=X|@C end
        proc {Pop X} S=@C in X=S.1 C:=S.2 end
        fun {IsEmpty} @C==nil end
in
        stack(push:Push pop:Pop isEmpty:IsEmpty)
end
```

- Any data abstraction can be implemented as an ADT or as an object

43

# Final remarks on objects

- Objects are omnipresent in computing today
- The first major object-oriented language was Simula-67, introduced in 1967
  - It directly influenced Smalltalk (starting in 1971) and C++ (starting in 1979), and through them, most modern object-oriented languages (Java, C#, Python, Ruby, and so forth)
- Most modern OO languages are in fact data abstraction languages: they incorporate both objects and ADTs
  - And other data abstraction concepts as well, such as components and modules
  - The Java language has both ADTs (e.g., Integer) and objects

44

# Four kinds of data abstraction

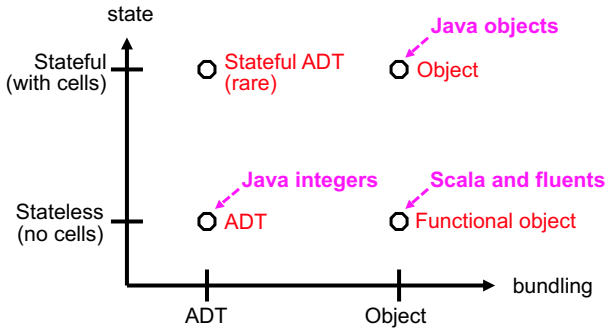
45

## Four kinds of data abstraction

- We have seen two kinds of data abstractions:
  - Abstract data types (without mutable state)
  - Objects (with mutable state)
- There are two other kinds of data abstractions
  - Abstract data types with state (stateful ADTs)
  - Objects without state (functional objects)
- This gives four kinds in all
  - Let's take a look at the two additional kinds
  - And then we'll conclude this lesson on data abstraction

46

# Four kinds of data abstraction




- Objects (with state) and ADTs (stateless) are in Java
- Functional objects are used in Scala and for big data
- Stateful ADTs are rarely used (so far!)

47

# The two less-used data abstractions

- A functional object is possible
  - Functional objects are immutable; invoking an object returns another object with a new value
  - Functional objects are becoming more popular
- A stateful ADT is possible
  - Stateful ADTs were much used in the C language (although without enforced encapsulation, since it is impossible in C)
  - They are also used in other languages (e.g., classes with static attributes in Java)
- Let's take a closer look at how to build them

48

# A functional object

- We can implement the stack as a functional object:

```
local
   fun {StackObject S}
      fun {Push E} {StackObject E|S} end
      fun {Pop S1}
         case S of X|T then S1={StackObject T} X end end
      fun {IsEmpty} S==nil end
   in stack(push:Push pop:Pop isEmpty:IsEmpty) end
in
   fun {NewStack} {StackObject nil} end
end
```

- This uses no cells and no secure wrappers. The simplest of all data abstractions since it only needs higher-order programming.

49

# Functional objects in Scala

- Scala is a hybrid functional-object language: it supports both the functional and object-oriented paradigms
- In Scala we can define an immutable object that returns another immutable object
  - For example, a RationalNumber class whose instances are rational numbers (and therefore immutable)
  - Adding two rational numbers returns another rational number
- Immutable objects are functional objects
  - The advantage is that they cannot be changed (the same advantage of any functional data structure)

50

# A stateful ADT

- Finally, let us implement our trusty stack as a stateful ADT:

```
local Wrap Unwrap
   {NewWrapper Wrap Unwrap}
   fun {NewStack} {Wrap {NewCell nil}} end
   proc {Push S E} C={Unwrap S} in C:=E|@C end
   fun {Pop S} C={Unwrap S} in
      case @C of X|S1 then C:=S1 X end
   end
   fun {IsEmpty S} @{Unwrap S}==nil end
in
   Stack=stack(new:NewStack push:Push pop:Pop isEmpty:IsEmpty)
end
```

- This uses both a cell and a secure wrapper. Note that Push, Pop, and IsEmpty do not need Wrap! They modify the stack state by updating the cell *inside* the secure wrapper.

51

# Conclusion

- Data abstractions are a key concept needed for building large programs with confidence
  - Data abstractions are built on top of higher-order programming, static scoping, explicit state, records, and secret keys
  - Data abstractions are defined precisely in terms of these concepts; our definitions give the semantics of data abstractions
- There are four kinds of data abstraction, along two axes: objects versus ADTs on one axis and stateful versus stateless on the other
  - Two kinds are more visible than the others, but the others also have their uses (for example, functional objects are used in Scala)
- Modern programming languages strongly support data abstractions
  - They support much more than just objects; it is more correct to consider them data abstraction languages and not just object-oriented languages

52

# LINFO1104
# Concepts, paradigms, and semantics of programming languages

## Lecture 8-9

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be



1

# Overview of lectures 8-9

- Refresher on semantics
  - We use the semantics to formally prove why tail recursion keeps the stack size constant
- 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



2

# Understanding tail recursion with semantics



3

# Using formal semantics to validate intuition

- We have seen the Oz language semantice
  - A mechanism called an abstract machine
  - Most popular languages can be defined with a similar abstract machine (Java, Scala, C++, C#, Python, etc.)
- In this course, we will occasionally use the semantics in order to explain a concept
  - Today we will explain tail recursion optimization (a.k.a. last call optimization) with the semantics



4

# Why does the tail-recursion rule work?

- We will use the semantics to show why stack size is constant when the recursive call is the last call
- We will use two versions of the factorial function, one with accumulator (Fact2) and the other without accumulator (Fact)
- We will execute both with the semantics
- These two examples generalize easily to any recursive function



5

# Factorial with accumulator

- Here is a (partial) translation of Fact2 into kernel language
  - (Why "partial"? There are three reasons!)

```
proc {Fact2 I A F}
   if I==0 then F=A
   else I1 A1 in
      I1=I-1
      A1=I*A
      {Fact2 I1 A1 F}
   end
end
```

- We will execute this definition with the semantics
- We will show that the stack size is the same just at each call to Fact2



6

# Start of Fact2 execution (1)

- Here is the instruction we will execute:

```
local N A F in
        N=5 A=1
        {Fact2 N A F}
end
```

- We suppose that the Fact2 definition is already in memory
- The actual instruction given to the abstract machine also contains the Fact2 definition (because memory is empty when the abstract machine starts)



7

# Start of Fact2 execution (2)

- Here is the complete instruction given to the abstract machine:
  - It executes with empty environment and empty memory

```
local Fact2 in
   proc {Fact2 I A F}
                if I==0 then F=A
                else I1 A1 in
                        I1=I-1
                        A1=I*A
                        {Fact2 I1 A1 F}
                end
   end
   local N A F in
                N=5 A=1
                {Fact2 N A F}
   end
end
```



8

# First call of Fact2

- Execution state at the first call:
  ([({Fact2 N A F}, {Fact2→*p*, N→*n*, A→*a*, F→*f* })],
  {*n*=5, *a*=1, *f*, *p*=(…)})
- After one step (execution of function body starts):
  ([(**if** I==0 **then** F=A **else** I1 A1 **in**
  I1=I-1 A1=A*I {Fact2 I1 A1 F} **end**,
  {Fact2→*p*,I→*n*,A→*a*,F→*f* })],
  {*n*=5, *a*=1, *f*, *p*=(…)})

- What is the contextual environment of Fact2?
- What is the environment for executing the function body of Fact2?



9

# Second call of Fact2

- Execution state at the second call:
  ([({Fact2 I1 A1 F},
  {Fact2→*p*, I→*n*, A→*a*, F→*f*, I1→$i_1$, A1→$a_1$ })],
  {*n*=5, *a*=1, $i_1$=4, $a_1$=5, *f*, *p*=(…)})
- You see that the stack only has one element
- It is easy to see that for each successive call to Fact2, the stack will only have one element
- QED!

- The book has a simpler example (Section 2.5.1)



10

# Factorial without accumulator

- Here is a (partial) translation of Fact into kernel language

```
proc {Fact N F}
   if N==0 then F=1
   else N1 F1 in
      N1=N-1
      {Fact N1 F1}
      F=N*F1
   end
end
```

- We will execute this definition with the semantics, with the call {Fact 5 F}
  - What is the complete instruction given to the abstract machine? (exercise)
- We will show that stack size increases by one element for each new recursive call



11

# Start of Fact execution

- Execution state at the first call of Fact:
  ([({Fact N F}, {Fact→*p*,N→*n*,F→*f* })],
  {*n*=5, *f*, *p*=(…)})
- Execution state at the **else** part of the **if** statement:
  ([(N1=N-1 {Fact N1 F1} F=N*F1,
  {Fact→*p*,N→*n*,F→*f*,N1→$n_1$,F1→$f_1$ })],
  {*n*=5, *f*, $n_1$, $f_1$, *p*=(…)})
  ← Contextual environment + arguments + 'local N1 F1'
- Execution state at the second call of Fact:
  ([({Fact N1 F1},
  {Fact→*p*,N→*n*,F→*f*,N1→$n_1$,F1→$f_1$ })
  (F=N*F1,
  {Fact→*p*,N→*n*,F→*f*,N1→$n_1$,F1→$f_1$ })],
  {*n*=5, *f*, $n_1$=4, $f_1$, *p*=(…)})
  ← Recursive call
  ← After the call



12

# Later in the execution

- One of the later calls to Fact:
  ([({Fact N1 F1}, {…}),
  (F=N*F1, {F→$f_2$,N→$n_2$,F1→$f_3$, …}),
  (F=N*F1, {F→$f_1$,N→$n_1$,F1→$f_2$, …}),
  (F=N*F1, {F→$f$,N→$n$,F1→$f_1$, …})],
  {$n$=5, $f$, $n_1$=4, $f_1$, $n_2$=3, $f_2$, …, $p$=(…)})
- At each new call, an instruction"F=N*F1" is put on the stack
  - The same instruction each time, but with a different environment!
- You can see that the stack stores all the multiplications that must be done at the end
  - When the base case is reached there is no more recursion, and all multiplications are executed



13

# Generalizing this result

- To prove this for all recursive functions, we need to define a schema for the execution of any recursive function
  - Schema = a representation of the set of all possible executions of all recursive functions
  - We redefine the semantics to work on the schema
  - This is not especially difficult, but it requires a bit of "theory bookkeeping"
- Does the stack grow for all non-tail-recursive functions?
  - Yes!
  - The complete formal verification of this fact is out of scope for this course, but if you are formally minded you can do it as an exercise (!)



14

# Conclusion

- When the recursive call is the last instruction in the body, the stack size is constant
- When the recursive call is not the last instruction, the stack size increases for each recursive call
  - The stack contains all instructions that must be executed later
- The semantics shows exactly what happens!
  - Our intuition on stack size is validated by the semantics



15

# Exceptions



16

# 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



17

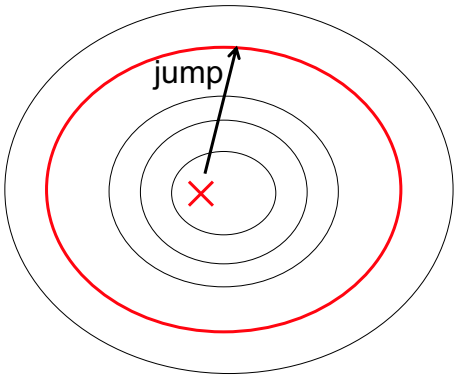
# The containment principle

- 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



18

# 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



19

# The try and raise instructions

- We introduce two new instructions for handling exceptions:

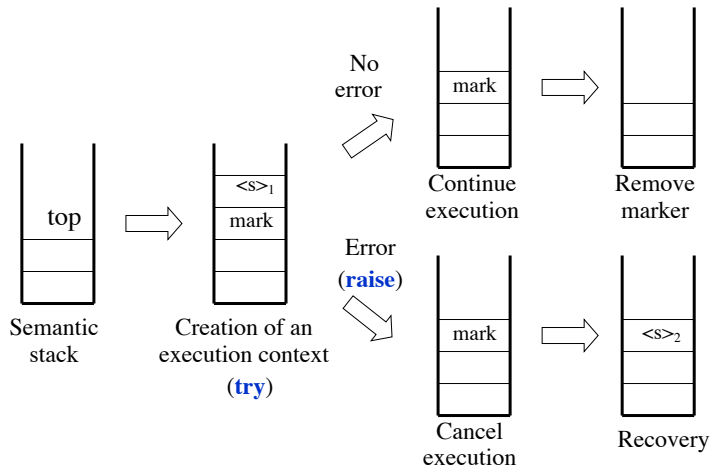
```
try <s>1 catch <y> then <s>2 end  % Create an execution context
raise <x> end                     % Raise an exception
```

- With the following behavior:
  - **try** puts a "marker" on the stack and starts executing $\langle s \rangle_1$
  - If there is no error, $\langle s \rangle_1$ 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 $\langle s \rangle_1$ is therefore canceled)
    - Then $\langle s \rangle_2$ is executed
    - <y> refers to the same variable as <x>
    - The scope of <y> exactly covers $\langle s \rangle_2$



20

# Semantics of exceptions



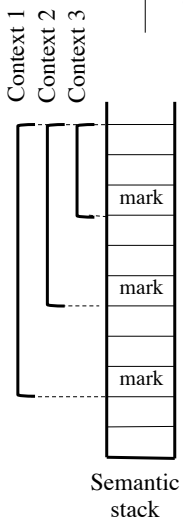



21

# An execution context

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

```
try                                  Context 1
   …                                 Context 2
   try                               Context 3
      …
      try
         …
         catch <x> then <s>3 end
      …
      catch <x> then <s>2 end
   …
   catch <x> then <s>1 end
```






22

# 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



23

# 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



24

## 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
```



25

## 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



26

# Java exception syntax

```
throw new NoSuchAttributeException(name);

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



27

# Good style

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

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



28

# 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!



29

# 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!



30

# Final remarks

- This completes the part of the course related to 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 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



31

# Concurrent programming



32

# 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 separate resources
  - **Operating system:** management of a single computer
    - A concurrent activity is called a process
    - Processes have independent memory spaces but share the same computer resources
  - **Process:** execution of a single program
    - A concurrent activity is called a thread
    - Threads share the same memory space



33

# 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
    - Communicate: information passes from one activity to another
    - Synchronize: an activity waits for another to perform a specific action



34

# *Concurrency can be (very) hard*

- It introduces many difficulties such as nondeterminism, race conditions, reentrancy, deadlocks, livelocks, fairness, handling shared data, and concurrent algorithms can be complicated
  - 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



35

# Deterministic dataflow



36

# 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 are going to 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



37

# 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?



38

# 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. A superb way to compute!



39

# 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?



40

# Continuing the execution

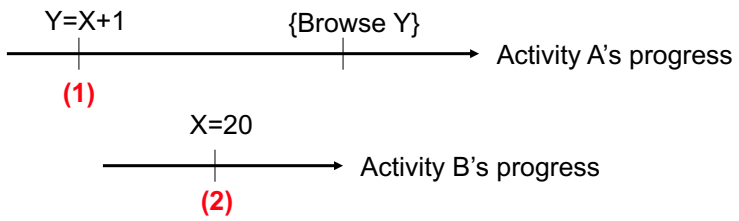
- The waiting instruction:
  ```
  declare X
  local Y in
      Y=X+1
      {Browse Y}
  end
  ```
- If someone would bind X, then execution could continue
- But who can do it?

- Answer: another concurrent activity!
- If another activity does:
  ```
  X=20
  ```
- Then the addition will continue and display 21!
- This is called dataflow execution



41

# 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



42

# 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



43

# 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!



44

# 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



45

# 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)



46

# 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



47

# 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 _ _ ]



48

# 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



49

# 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]



50

# 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 targets the dataflow paradigm
  - The Browser does not look at the content of cells, since they do not execute with dataflow



51

# Streams and agents



52

# 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



53

# 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!



54

# Producer/ consumer (1)

- A producer generates a stream of data

```
fun {Prod N} {Delay 1000} N|{Prod N+1} end
```

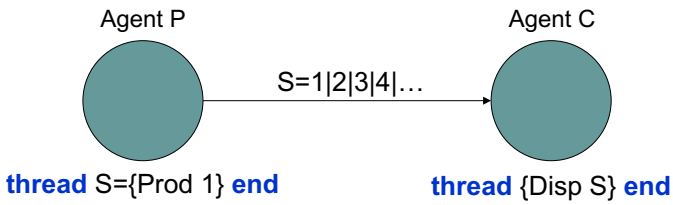
  - The {Delay 1000} slows down execution enough to observe it
- A consumer reads the stream and performs some action (like the Disp procedure)
- A producer/consumer program:

```
declare S
thread S={Prod 1} end
thread {Disp S} end
```



55

# 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



56

# 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
```

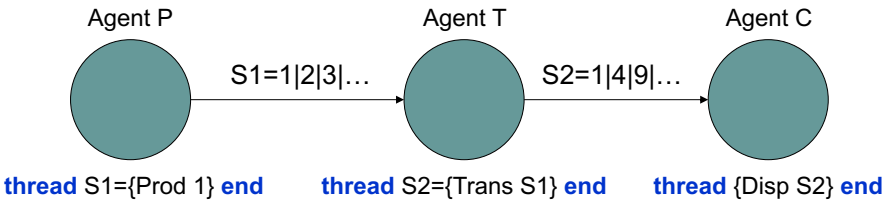
- This program has three agents:

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



57

# 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



58

# 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 makes 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! In the next lesson will see more examples of the power of the model.



59

# Thread semantics



60

# Thread semantics (1)

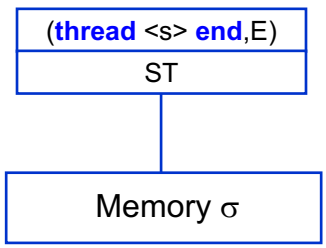
- We extend the abstract machine with threads
- Each thread has one semantic stack
  - The instruction **thread** <s> **end** creates a new stack
  - All stacks share the same memory
- There is one sequence of execution states, and threads take turns executing instructions
  - $(MST_1,\sigma_1) \rightarrow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



61

# Thread semantics (2)

A semantic stack that is about to create a thread

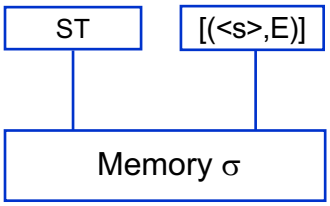





62

# Thread semantics (3)

We now have two stacks!






63

# Why interleaving semantics?

- What happens when activities execute ”at the same time”?
- We can imagine that all threads execute in parallel, each with its own processor but all sharing the same memory
  - We have to be careful to understand what happens when threads operate simultaneously on the same memory word
  - If the threads share the same processor, then this problem is avoided (interleaving semantics)
- Interleaving semantics is much easier to reason about than true concurrency semantics
  - 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



64

# Order of execution states



65

# Order of execution states

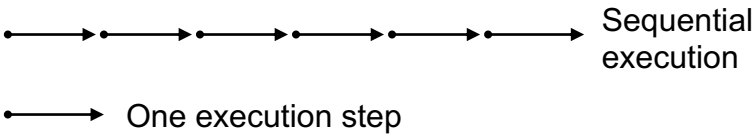
- In a sequential program, execution states are in a total order
  - Total order = when comparing any two execution states, one must happen before the other
- In a concurrent program, execution states of the same thread are in a total order
  - The execution states of the complete program (with more 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)



66

# Total order of a sequential program

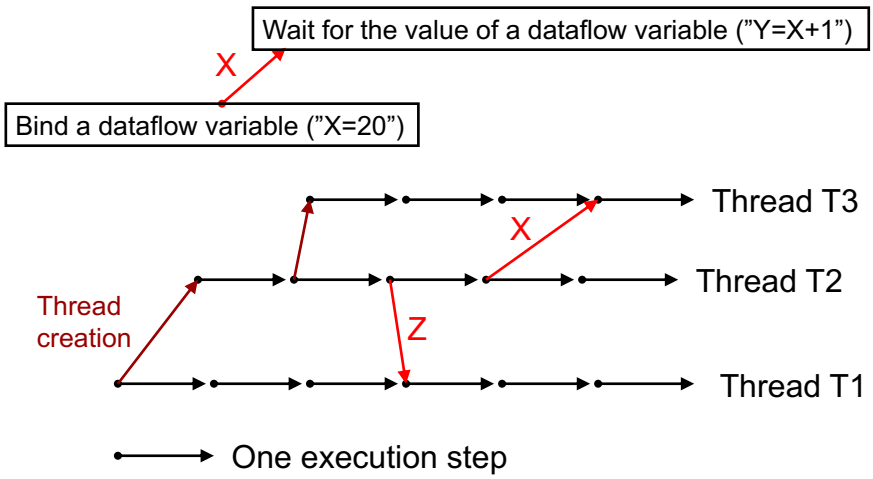
- In a sequential program, execution states are in a total order
- A sequential program has one thread






67

# Partial order of a concurrent program

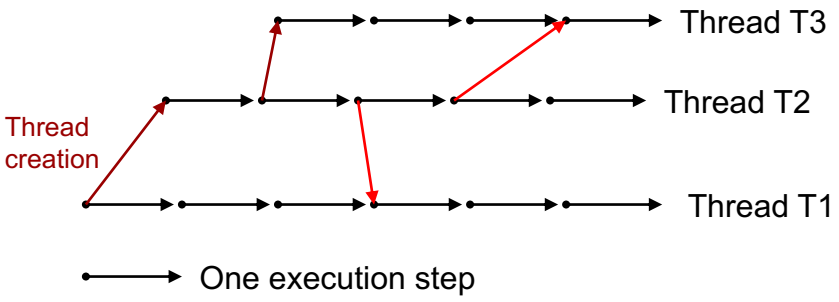





68

# 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)

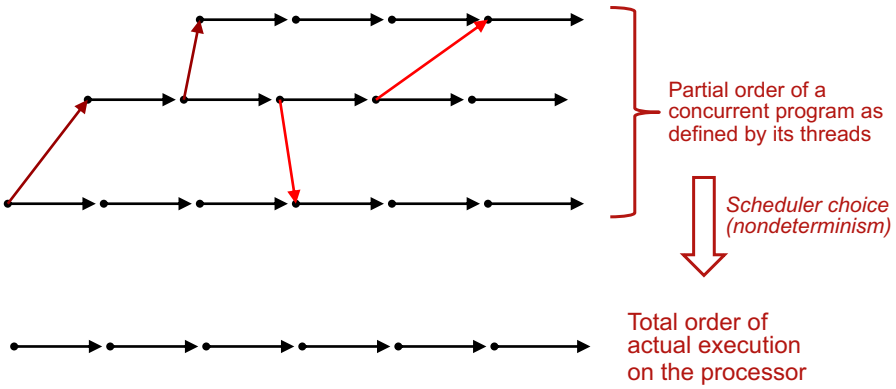





69

# The actual execution order

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






70

# Nondeterminism



71

# 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



72

# 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



73

# 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



74

# 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
- 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





# 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

# 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.



77

# 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



78

# “Concurrency for dummies”



79

# “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!)



80

## Example (1)

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



81

## Example (2)

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



82

# 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



83

# 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}}
  ```



84

# 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.



85

# 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`



86

# 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]



87

# 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.



88

# 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
```



89

# 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
```

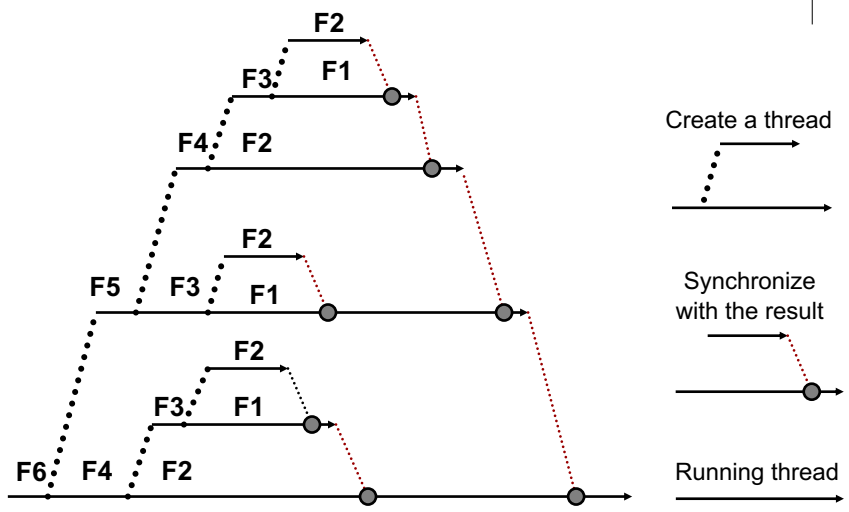
Dataflow dependency

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



90

# Execution of {Fib 6}

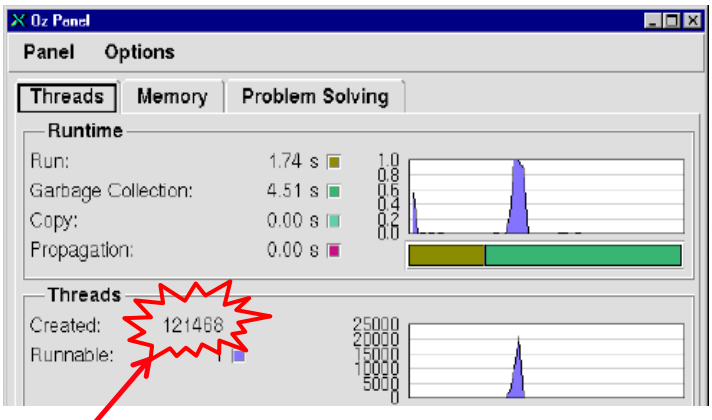





91

# Observing the execution of Fib

Only in Mozart 1




Total number of threads created since system startup

Oz Compiler Panel (in Oz menu)



92

# Counting threads

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

This works also in Mozart 2



93

# Multi-agent programming



94

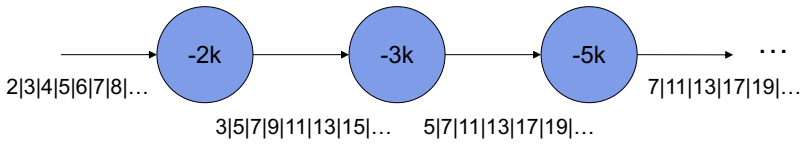
# Multi-agent programming

- 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



95

# 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



96

# 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
```



97

# 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)



98

# 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?)



99

# Digital logic simulation



100

# 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



101

# 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



102

# Digital signals as streams

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

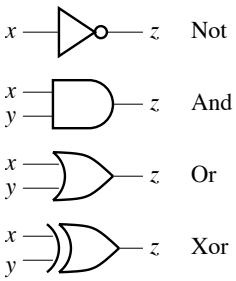
  $S=a_0|a_1|a_2|...|a_i|...$

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



103

# Digital logic gates




| | | z | | | |
|---|---|---|---|---|---|
| x | y | 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!



104

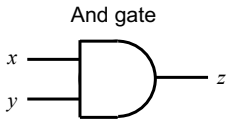
# Digital gates as agents

- 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
```






105

# Creating many gates

- Let us define a proper 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



106

# 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
```



107

# 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}
```



108

# Combinational logic

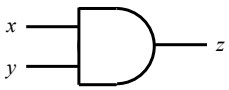


109

# Combinational logic

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

$z_i$ = $x_i$ And $y_i$

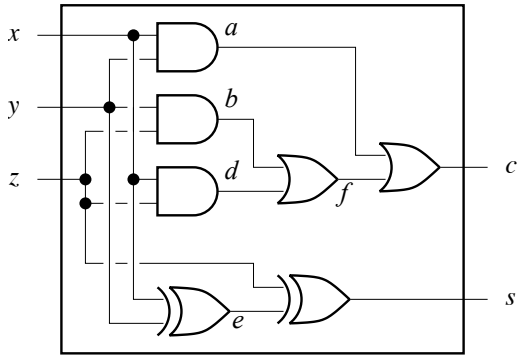



- Therefore, any number of interconnected gates also defines a combinational function
- We define a useful circuit called a full adder



110

# 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



111

# 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
```



112

# Sequential logic

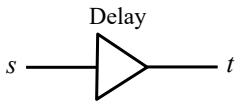




## Sequential logic

- 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

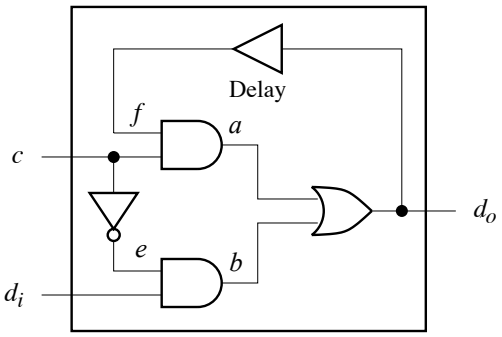
$S=a_0|a_1|a_2|...|a_i|...$
$T=b_0|b_1|b_2|...|b_i|...$




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

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

# 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



115

# 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
```



116

# Summary and history



117

# 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 »



118

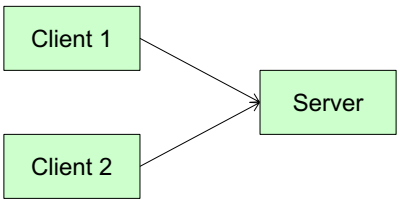
# Historical note: concurrency *must* get simpler

- 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



119

# 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

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

- 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



120

# 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).

# LINFO1104
# Concepts, paradigms, and semantics of programming languages

## Lecture 10-11

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.
- Message-passing concurrency (multi-agent actor programming)
  - We overcome the limitation by adding one new concept, ports, to deterministic dataflow
  - We define port objects and active objects and show how to write message protocols and multi-agent actor programs
  - We show another approach using ports, namely use deterministic dataflow by default and add ports but only when necessary

2

# Limitation of deterministic dataflow

3

## Limitation 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.

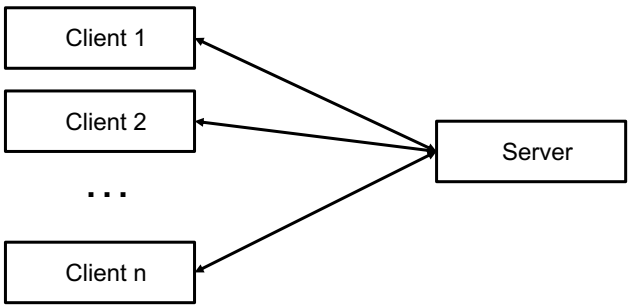
4

# 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)

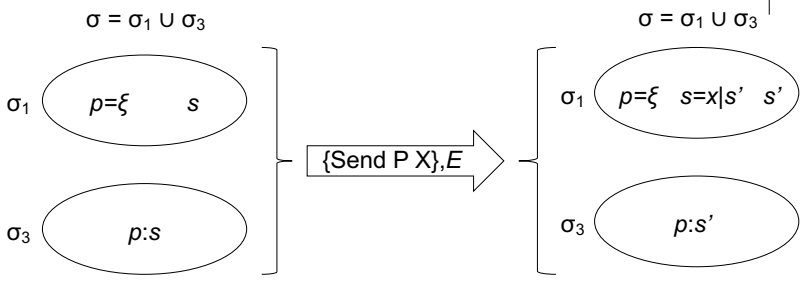
- Assume single-assignment store $\sigma_1$ with variables
- Assume a port store $\sigma_3$ that contains pairs of variables
  - (Remember $\sigma_2$ was the cell store we introduced before)

environment

- P={NewPort S}, {P→*p*, S→*s*}
  - Assume unbound variables *p*, *s* ∈ $\sigma_1$
  - Create fresh name ξ, bind *p*=ξ, add pair *p*:*s* to $\sigma_2$
- {Send P X}, {P→*p*, X→*x*}
  - Assume *p*=ξ, unbound variable *s* ∈ $\sigma_1$ , *p*:*s* ∈ $\sigma_2$
  - 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 will show how to write concurrent programs in this new paradigm
  - We will define port objects and active objects
  - We show how to do message protocols
  - We show how to combine deterministic dataflow with ports, leading to the best all-round paradigm for concurrent programming

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
```

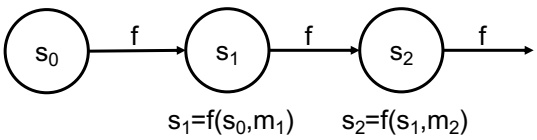
Using syntax of **for** loops

30

# Stateful port objects (stateful agents)

31

## Stateful port objects (Section 5.2)




- 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: State × Msg ↦ 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
```

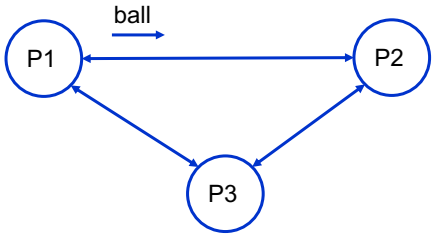
- P is hidden inside the procedure by lexical scoping
- This makes it easier to use port objects or standard objects as we saw before

41

## Play ball example

42

# Play ball example



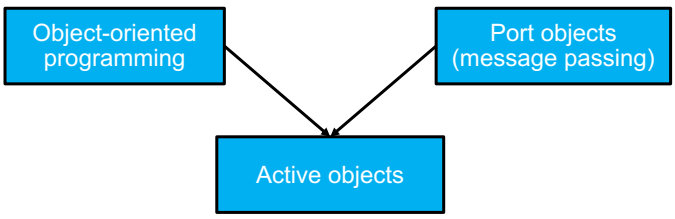

- This is a simple multi-agent program using stateful port objects
  - Three players stand in a circle. There is one ball. A player who receives the ball will send it to one of the other two, chosen randomly.
  - Each player counts the number of times it has received the ball, and it responds to a query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

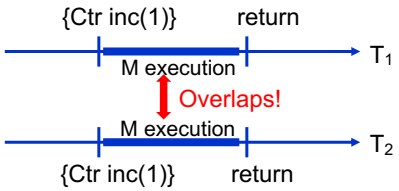
47

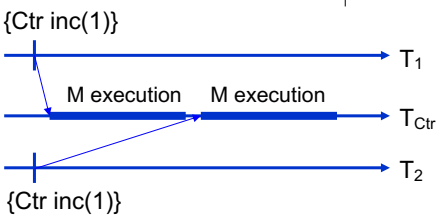
# Passive objects and active objects

- 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

49

# 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

50

# Message protocols

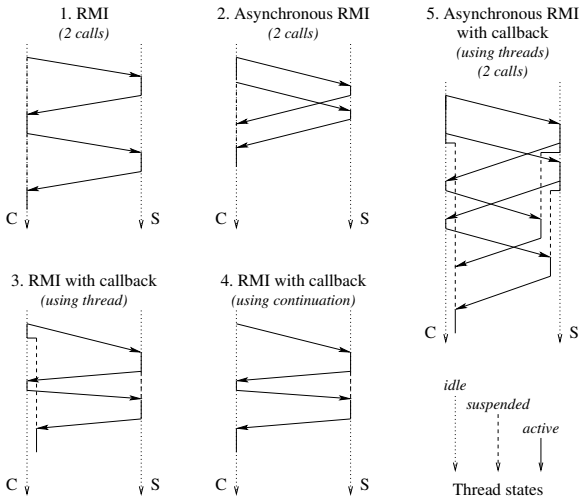
51

## 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:
  ([({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}) →
  …
- 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

{$i_0$,…,$i_8$} not needed!
Only $i_9$ is needed (from stack)

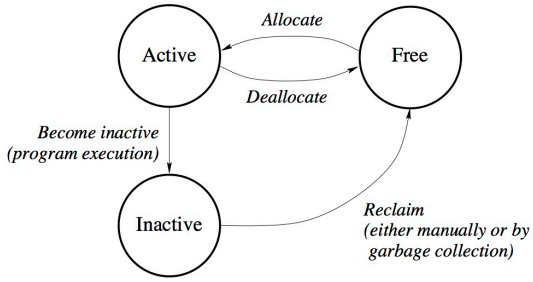
57

# 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 $\sigma_1$ never shrinks: we only add new variables
    - The same thing happens to the cell store $\sigma_2$: 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

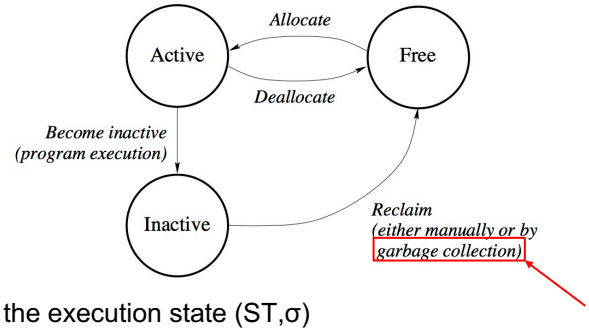
63

# 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 $\sigma_1$ is the variable store and $\sigma_2$ is the cell store
  - Denote by $V_\sigma$ 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:
  - $x_0 \in V_{ST}$
  - $1 \le \forall i < n : x_{i-1} \longmapsto x_i$ 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

65

# Execution of {Loop10 0} (3)

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

66

# 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 help with general programs?
- 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



# 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

# 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

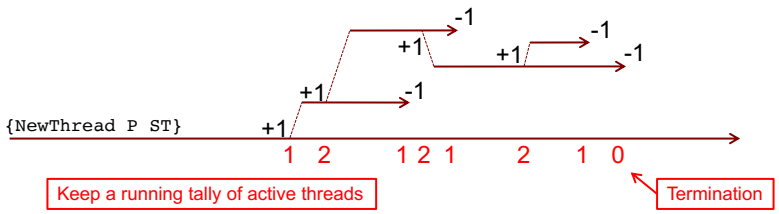
- The main thread waits until all subordinate threads are terminated
- We can define this abstraction as follows:

```
{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
   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
```

Done inside the new thread

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!



# 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

# 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 ways to build 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!)

# Introduction to Resilient Distributed Programming in Erlang

LINFO1104 and LINGI1131
May 12 and May 13, 2020

Peter Van Roy
Université catholique de Louvain

This lecture is based on information taken from many Erlang documents
Prerequisites for the lecture are some knowledge of functional programming and message passing

1

## Overview

- Erlang performance
  - Some numbers to show Erlang's strengths
- Basic concepts of the Erlang language
  - Pure functional core with dynamic typing
  - Process (agents) and message passing with mailboxes
  - Failure detection with linking and monitoring
  - Dynamic code updating
- Programming abstractions of the Erlang/OTP platform
  - Basic principles of resilient system design
    - Erlang stable storage: ETS, DETS, Mnesia
    - Testing tools: EUnit, Common Test, Dialyzer
  - Standard behaviors (concurrency patterns)
    - Generic servers, generic FSMs, generic event handling, supervisor trees
    - Principles of a generic server
    - Principles of supervisor trees
- Conclusions

2

# Erlang introduction

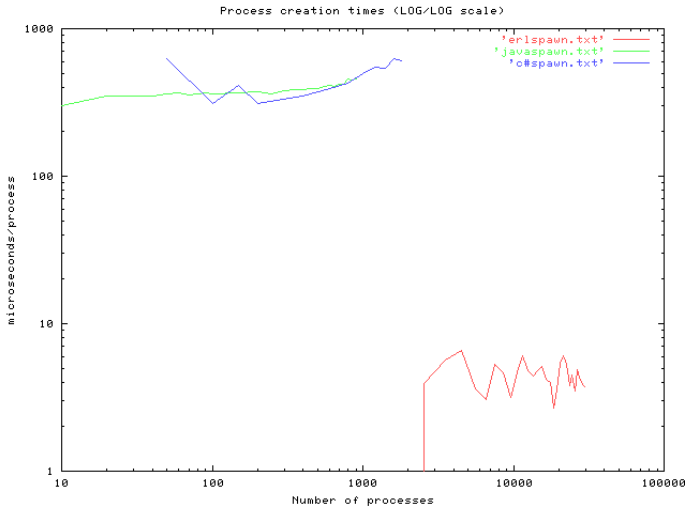
- Erlang was developed in Ericsson for telecommunications in 1986 (Java is from 1991)
  - It is released as OTP (Open Telecom Platform) with a full set of libraries
  - It is supported by Ericsson, the Erlang Ecosystem Foundation, and a user community (www.erlang.org)
- Erlang programs consist of lightweight "processes", which are active agents that communicate using asynchronous FIFO message passing ("actor model" as proposed by Carl Hewitt in 1973)
  - 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/OTP supports reliable long-lived distributed systems using behaviors and supervisors
  - Behaviors are generic concurrency patterns that make it easy to write robust concurrent systems
  - Supervisors are a general pattern to observe processes and restart them when they crash
  - 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!)

3

# Erlang performance

4

# Erlang process creation times

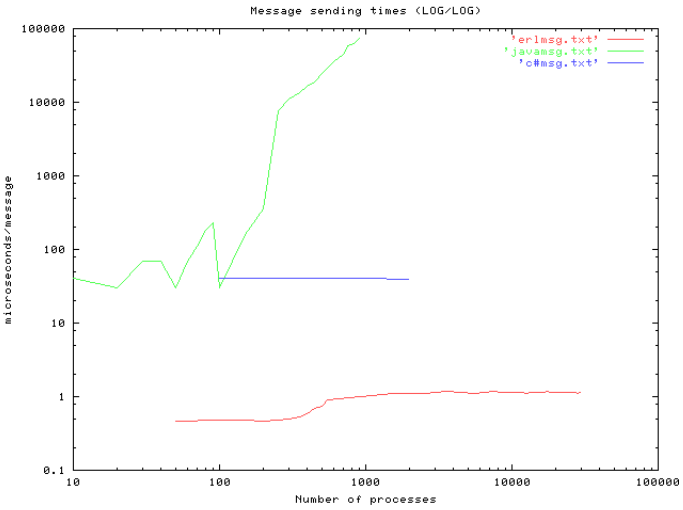



From Joe Armstrong, Concurrency Oriented Programming in Erlang, Nov. 2002.

- We compare process creation times for Erlang, Java, and C#
- These numbers were measured in 2002

5

# Erlang message sending times

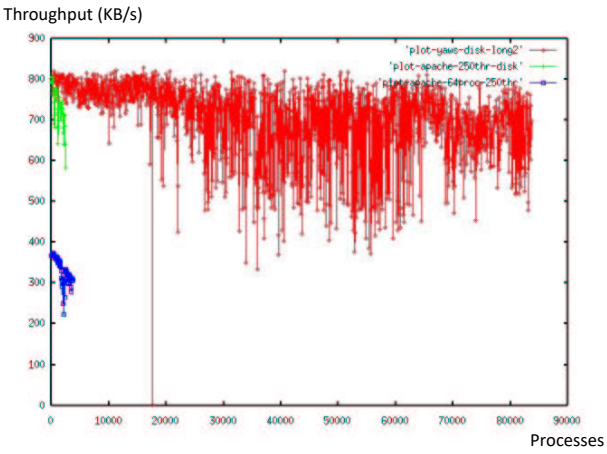



From Joe Armstrong, Concurrency Oriented Programming in Erlang, Nov. 2002.

- We compare message sending times for Erlang, Java, and C#
- These numbers were measured in 2002

6

# Use case: web server

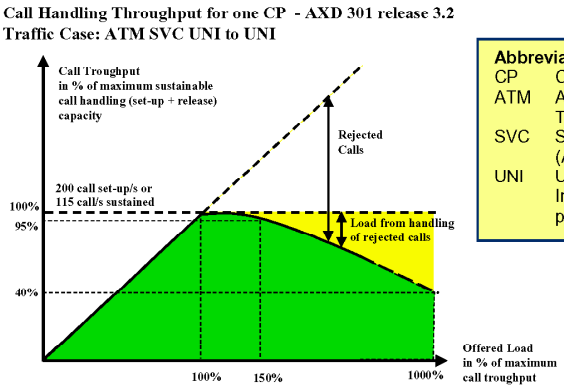



From Joe Armstrong, Concurrency Oriented Programming in Erlang, Nov. 2002.

- Throughput versus number of processes for Web servers
  - Red = yaws (Yet Another Web Server, in Erlang on NFS)
  - Green = apache (local disk)
  - Blue = apache (NFS)
- Yaws: 800 KB/s up to 80,000 processes
- Apache: crashes at around 4,000 processes

7

# Use case: AXD301 Erlang-based ATM switch




**Abbreviations:**

| | |
|---|---|
| CP | Control Processor |
| ATM | Asynchronous Transfer Mode |
| SVC | Switched Virtual (ATM) Channel |
| UNI | User Network Interface signaling protocol |

From Ulf Wiger, Four-fold Increase in Productivity and Quality, 2001.

- The AXD 301 is a general-purpose high-performance ATM switch from Ericsson
  - The AXD 301 is built using Erlang OTP supplemented with C and Java
- Throughput drops linearly when overloaded
  - 95% throughput at 150% load, descending to 40% throughput at 1000% sustained load
- AXD 301 release 3.2 has 1MLOC Erlang, 900KLOC C/C++, 13KLOC Java
  - In addition, Erlang/OTP at that time had 240KLOC Erlang, 460KLOC C/C++, 15KLOC Java

8

# Basic concepts

9

## Pure functional core

- Within a process, Erlang runs as a pure functional language
  - All variables are single assignment (bound when they are declared)
  - Functions are values with lexically scoped higher-order programming
  - Pattern matching can be used in **case** and **if** statements (and **receive**)
- All data structures are symbolic values
  - Integers (arbitrary precision), floats, atoms (symbolic constants)
  - Lists [george,paul,john,ringo] and tuples {Key,Val,L,R}
  - Strings are lists of ASCII codes (integers)
  - Binary vectors (used for protocol computations)

10

# Dynamic typing versus static typing

- Erlang is a strongly typed language, that is, types are enforced by the language
  - Many popular languages are strongly typed, such as Java, Scheme, Haskell, and Prolog
  - Weakly typed languages, e.g., C and C++, allow access to a type's internal representation
- Strongly typed languages can be dynamically or statically typed
  - Erlang is a dynamically typed language because variables can be bound to entities of any type
  - In a statically typed language, variable types are known at compile time
- Static typing allows catching more program errors at compile time
  - However, this does not mean that statically typed programs are more resilient
    - Static typing allows catching many "surface errors" but does not help with "deep errors"
  - Well-written Erlang programs are among the most resilient software artefacts ever built, because Erlang provides adequate mechanisms to overcome deep errors

11

# An Erlang module

- The source code of an Erlang program is organized in modules:

```
-module(math).
-export([areas/1]).
-import(lists, [map/2]).

areas(L) -> lists:sum(map(fun(I) -> area(I) end, L)).

area({square,X}) -> X*X;
area({rectangle,X,Y}) -> X*Y.
```

- Modules import and export, giving a dependency graph of modules

12

## Creating processes and sending messages

- Any process can create another by calling spawn
  - Pid = spawn(Fun) : function Fun defines the behavior, Pid is the process name
  - Fun may be anonymous or named
    - `fun(args) -> expr end`
    - `fun name/arity`
- The process name Pid is a unique constant that identifies the process
- Messages can be sent to the process using the process name
  - Pid ! Message
  - Messages are sent asynchronously and all data in messages is copied
  - Messages can be received by the **receive** statement

13

## Receiving messages

- Each process has a mailbox that contains an ordered list of messages received by the process
- Messages are extracted from the mailbox using the **receive** statement
  - The **receive** uses pattern matching to remove the first message that matches
  - **receive**
    pattern1 **when** guard1 -> expr1;
    pattern2 **when** guard2 -> expr2;
    …
    patternN **when** guardN -> exprN
    **end**

14

# Send and receive

```
Pid ! Message,
…

receive
      Message1 ->
       Actions1;
      Message2 ->
       Actions2;
      …
      after Time ->
       TimeOutActions
end
```

15

# Receive mailbox semantics

- When a process executes **receive**:
  - If the mailbox is empty, the **receive** blocks and waits for a message
  - If the mailbox is not empty, it takes the first message and tries patterns in order starting from the first, if it finds a matching pattern it executes the corresponding code
  - If no pattern matches, the **receive** blocks and waits for the next message
    - Unmatched messages remain in the mailbox and can be removed by future **receive** calls
    - This allows different parts of a process to treat different kinds of messages
    - Messages can be removed out-of-order (in a different order from when they arrived)
    - Care must be taken that messages do not stay in the mailbox forever (memory leak)
- Patterns are symbolic data structures containing variable identifiers and guards are simple built-in tests

16

## Process registering

- A process Pid can be registered with a name, which is an atom, to make the process globally available
  - This is important to keep interfaces unchanged, even as processes crash and are restarted
- register(Atom, Pid) : give Pid the global name Atom
- unregister(Atom) : remove the registration for Atom
- whereis(Atom) -> Pid | undefined : returns the Pid of a registered process, or undefined if no such process exists
- registered() -> [Atom :: atom()] : returns a list of all registered processes

17

## Process linking

- Two processes can be linked together
  - Process Pid1 calls link(Pid2) or conversely; linking is bidirectional
- Process termination: send exit signal
  - A process terminates with an exit reason, which is sent as a signal to all linked processes
  - When a process terminates normally, the exit reason is the atom `normal`, otherwise when there is a run-time error, the exit reason is {Reason,Stack}
- Propagating process termination: transitive by default ("link set")
  - The default behavior when a process receives an exit signal with reason other than normal is to terminate and to send exit signals with the same reason to its linked processes
  - A process can be set to trap exit signals by calling process_flag(trap_exit,true)
  - A received exit signal is then transformed into the message {'EXIT', FromPid, Reason}, which is put into the mailbox of the process

18

# Using process linking

- If a process throws an exception that is not caught at the top level, then the process terminates and broadcasts its exit signal to all linked processes
- A few additional operations are defined to manage this:
  - exit(Pid,Why): send an exit signal to Pid without terminating
  - exit(Pid,kill): send an unstoppable exit to process Pid

```
start() -> spawn(fun go/0).

go() ->
     process_flag(trap_exit, true),
     loop().

loop() ->
     receive
       {'EXIT',Pid,Why} -> …
     end.
```

19

# Process monitoring

- Process monitor is an asymmetric version of linking
  - No resemblance to monitor concept in shared-state concurrency!
- For example, in a client/server, if the server crashes we want to kill the clients, but if a client crashes we do not want to kill the server
- If process Pid1 executes:

  Ref = erlang:monitor(process, Pid2)

- Then if Pid2 dies with exit reason Why, then Pid1 will be sent a message {'DOWN',Ref,process,B,Why}

20

# Distributed Erlang

- A distributed Erlang system consists of a number of Erlang runtime systems, called *nodes*, communicating with each other
- Message passing between processes at different nodes, as well as links and monitors, is transparent when using Pids
- Nodes can spawn processes on other nodes using spawn(Node,M,F,A)
- Registered names are local to each node: both node and name must be specified when sending messages using registered names
- The first time the name of a node is used, a connection attempt is made to the node; connections are made transitively giving a fully connected mesh by default (recent versions of Erlang allow more scalable connection topologies)

21

# Dynamic code change (1)

- In a real-time system, we would like to change the code without stopping the system
  - Some systems are never supposed to be stopped, e.g., the X2000 satellite control system developed by NASA
  - Hot code changing is difficult in a monolithic programming system, however, Erlang makes it possible because processes are independent (no sharing)
- Erlang allows for each module to have two versions of code
  - All new processes will be dynamically linked to the latest version
  - If the code is changed, then processes can choose to continue with the old code or to use the new code
  - The choice is determined by how the code is called

22

# Dynamic code change (2)

- Call the new version (if available)

```
-module(m).

loop(Data, F) ->
     receive
     (From,Q} ->
          {Reply,Data1}=F(Q,Data),
          m:loop(Data1, F)
     end.
```

Use new version

- Keep calling the old version:

```
-module(m).

loop(Data, F) ->
     receive
     {From,Q} ->
          {Reply,Data1} =F(Q,Data),
          loop(Data1, F)
     end.
```

Use old version

- This mechanism is used by libraries that manage upgrading of application releases

23

# Client/server process with hot code swap

- Dynamic code change can also be done for individual processes using higher-order functions

Process code

```
server(Fun, Data) ->
     receive
       {new_fun,Fun1} ->
           server(Fun1,Data);
       {rpc,From,ReplyAs,Q} ->
           {Reply,Data1} =
                Fun{Q,Data},
           From!{ReplyAs,Reply},
           server(Fun, Data1)
     end.
```

```
rpc(A,B) ->
     Tag=new_ref(),
     A!{rpc,self(),Tag,B},
     receive
        {Tag,Val} -> Val
     end.
```

24

# Programming abstractions

25

## Erlang philosophy

- How can we make robust software?
  - Popular languages (e.g., Java and Python) are inadequate
- Principles of robust software (from Joe Armstrong's Ph.D. thesis)
  - Errors cannot be fully eliminated, therefore they must be handled (both hardware and software errors)
  - Software components are the units of failure: errors occurring in one will not affect others ("strong isolation")
  - Software should be fail-fast: either function correctly or stop quickly
  - Failure should be detectable by remote components
  - Software components share no state, but communicate through messages

26

# Erlang “slogans”

- “Let it crash”, “If you can’t do your job, crash”
  - Instead of trying to fix things when errors happen, which leads to a large number of complicated states, instead map everything to one simple state, namely “crashed”
- “Let some other process do error recovery”
  - Both hardware and software errors can occur and trying to solve them in the process makes things complicated, better to detect and handle any error, hardware or software, elsewhere
- “Do not program defensively”
  - Defensive programming means to add checks in the program. This is not productive since it makes the program complicated (in particular, what do you do when a check fails?) and it will not remove all errors. Errors will still occur and still need to be handled. The best way is to map all errors no matter how bizarre to a single fault state, namely “crashed”.

27

# Erlang/OTP systems

- Erlang/OTP supports a hierarchy of systems:
  - Release: Contains all the information necessary to build and run a system, including a software archive and a set of procedures for installation (including upgrading without stopping)
  - Application: Contains all the software necessary to run a single application, not the entire system. Releases are often composed of multiple applications that are largely independent of one another, or that are hierarchically dependent.
  - Behavior: A set of processes that together implement a concurrency pattern
    - A notable behavior is supervisor: a tree of processes whose purpose is to monitor behaviors and each other and restart them when necessary
  - Worker: A process that is an instance of a behavior, usually instances of gen_server, gen_event or gen_fsm

28

# Using behaviors to abstract concurrency and fault tolerance

- In general, program code can be structured into “difficult” and “easy” modules
  - The difficult modules should be few and written by expert programmers
  - The easy modules should be many and written by regular application programmers
- Concurrency and fault tolerance are difficult to implement correctly
  - Erlang provides behaviors, generic components to hide concurrency and fault tolerance
  - Behaviors are Erlang’s “programming patterns” for building robust concurrent programs
- The Erlang/OTP platform provides library support for many important behaviors
  - See Erlang/OTP System Documentation, Ericsson, Version 10.7, March 15, 2020

29

# Standard Erlang/OTP behaviors

- The Erlang/OTP platform provides the following standard behaviors:
  - Generic server (gen_server): to build client/server architectures with registration, start/stop, timeouts, state management, synchronous/asynchronous calls, error handling
  - Generic event handler/manager (get_event): event handlers, such as loggers, to respond to a stream of events, handling them and sending notifications
  - Generic finite state machine (gen_fsm): applications (e.g., protocol stacks) can be modeled as finite state machines, which provides a set of rules State × Event → Actions × State
  - Application: a component that can be started and stopped as a unit, and can be reused in other systems
  - Supervisor: the generic toolkit for implementing supervisor hierarchies
- These behaviors hide most of the complexity of each concept, in particularly both concurrency and fault tolerance are vastly simplified using behaviors
  - All non-supervisor behaviors are designed to be pluggable into a supervisor hierarchy

30

# Erlang stable storage

- Resilient systems need stable storage to survive crashes
  - When a process is restarted by a supervisor, it uses the stable storage to start in a consistent state
  - Stable storage and supervisor trees are the two pillars of Erlang's resilience
- Erlang provides three levels of stable storage
  - ETS (Erlang Term Storage): Efficient in-memory storage for arbitrary Erlang terms, with limited size and concurrency abilities
  - DETS (Disk-based ETS): Same abilities as ETS, but stored on disk
    - ETS and DETS are limited to single nodes (single Erlang virtual machine)
  - Mnesia: A transactional database that is built on top of ETS and DETS and allows making a balance between ETS efficiency and DETS persistence
    - Mnesia supports distribution and replication on multiple nodes

31

# Testing

- It's not enough to design for resilience, testing is essential
- Erlang provides a spectrum of practical and powerful testing tools
  - EUnit: Unit testing framework, including test generators (using higher-order programming to generate new tests) and fixtures (scaffolding around tests)
    - Supports test-driven development (TDD)
    - Good for testing modules and libraries
  - Common Test: Full-featured system testing framework
    - Test groups: allows running tests in parallel or in random order, for race conditions
    - Test suites: for handling dependencies between applications when testing
    - Test specifications including simulating of abnormal termination (fault injection)
  - Dialyzer: Dynamic type checker based on success types
    - Will not make a proof of correctness, but any type error it finds is a real error

32

# Generic servers

33

# Generic server example

- Client/server code can be written as a generic part plus a specific part
- Consider the following simple (non-generic) server, which keeps track of "channels" that can be allocated and deallocated

```
-module(ch1).
-export([start/0]).
-export([alloc/0,free/1]).
-export([init/0]).

start() -> spawn(ch1, init, []).

alloc() -> ch1 ! {self(),alloc},
     receive {ch1, Res} -> Res end.

free(Ch) -> ch1 ! {free, Ch}, ok.
```

```
init() ->
     register(ch1, self()),
     Chs=channels(),
     loop(Chs).

loop(Chs) ->
     receive
        {From,alloc} ->
            {Ch, Chs2} = alloc(Chs),
             From!{ch1,Ch},
             loop(Chs2);
        {free,Ch} ->
             Chs2=free(Ch, Chs),
             loop(Chs2)
     end.
```

34

# Generic server: generic part

- The server code can be rewritten as a generic part plus a callback module
- We give first the generic part

```
-module(server).
-export([start/1]).
-export([call/2,cast/2]).
-export([init/1]).

start(Mod) ->
    spawn(server,init,[Mod]).
call(Name,Req) ->
    Name ! {call, self(), Req},
    receive {Name,Res} -> Res end.
cast(Name,Req) ->
    Name ! {cast, Req}.
```

```
Init(Mod) ->
     register(Mod, self()),
     State=Mod:init(),
     loop(Mod, State).

loop(Mod, State) ->
     receive
       {call,From,Req} ->
          {Res,State2}=
            Mod:handle_call(Req,State),
          From ! {Mod, Res},
          loop(Mod, State2);
       {cast, Req} ->
          State2 =
            Mod:handle_cast(Req,State),
          loop(Mod, State2)
     end.
```

35

# Generic server: callback module

- Here is the callback module
  - This is the only thing the programmer needs to write
  - Server name ch2 and protocol messages are hidden from clients

```
-module(ch2).
-export([start/0]).
-export([alloc/0,free/1]).
-export([init/0,
         handle_call/2,
         handle_cast/2]).
```

```
start() -> server:start(ch2).
alloc() ->
   server:call(ch2, alloc).
free(Ch) ->
   server:cast(ch2,{free,Ch}).
```

```
init() -> channels().
handle_call(alloc, Chs) ->
     alloc(Chs).
handle_cast({free, Ch}, Chs) ->
     free(Ch, Chs).
```

Connection between generic part (init, handle_call, handle_cast) and specific part (channels, alloc, free)

36

# Channels implementation

- Here are channels, alloc, and free as used by the gen_server example
  - Channels manage a 2-tuple {A,F} where A is allocated channels and F is free channels

```
channels() -> {_Allocated=[], _Free=lists:seq(1,100)}.

alloc({Allocated, [H|T]=_Free}) -> {H, {[H|Allocated], T}}.

free(Ch, {Alloc, Free} = Channels) ->
     case lists:member(Ch, Alloc) of
        true ->
           {lists:delete(Ch, Alloc), [Ch|Free]};
        false ->
           Channels
     end.
```

37

# Generic server behavior

- The Erlang/OTP gen_server extends the behavior of this example
- Starting a gen_server
  - The name of the callback module is specified
- Requests
  - Synchronous requests (call): there is a return value
  - Asynchronous requests (cast): no return value
- Stopping a gen_server
  - If the gen_server is part of a supervision tree, no stop function is needed. The server will automatically be terminated by its supervisor.
  - If the gen_server is standalone, a stop function may be useful

38

# Supervisor trees

39

## Supervisor trees introduction

- In practical concurrent and distributed systems, it is observed that most faults and errors are transient
  - For example: network problems, timing problems, concurrent startup
  - Simple retrying is a surprisingly successful strategy
- Supervisor trees are designed to favor this strategy
  - Process sets are “supervised” (observed for failure) by supervisor processes
  - Supervisors have authority to stop and restart supervised processes
  - Supervisors are themselves observed, in case they fail
- Supervisor trees are carefully implemented to avoid races
  - Starting of a supervisor tree is synchronous, to establish correct initial state

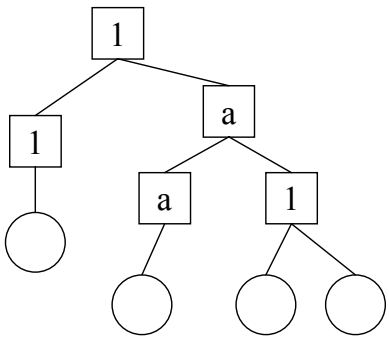
40

# Supervisor structure and principles

- Supervisor tree is a hierarchy with a root
  - A supervisor tree consists of a set of supervisor nodes, organized as a hierarchy with a root node and internal nodes (the root is a very important process!)
  - A supervisor node is responsible for starting, stopping, and monitoring its child processes
  - All Erlang behaviors are designed to work together with supervisors
- Restart principles
  - Restart strategy: one_for_one, one_for_all, rest_for_one
  - Restart frequency: the number of restarts is limited per time interval
    - If the limit is exceeded, the supervisor terminates and the next higher level supervisor takes some action
    - The intention is to prevent a situation where a process dies repeatedly for the same reason and is always restarted

41

# Supervision hierarchies

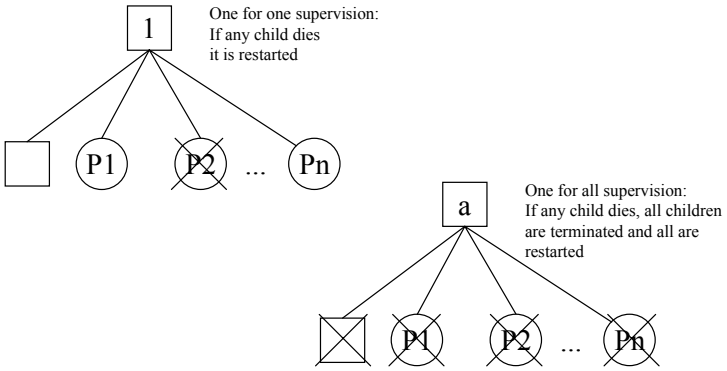



**Abbreviations:**
a One-for-all supervision
1 One-for-one supervision

- A supervisor (□) is a process whose sole purpose is to start, monitor, and possibly restart workers (○)
- A worker is an instance of a behavior, which raise exceptions when errors occur

42

# One-for-one and one-for-all supervision




- Different forms of supervision depending on how the children processes work
  - One-for-one: if the children are independent (each manages one connection)
  - One-for-all: if the children are collaborating, then if one crashes they all have to be restarted, even the correct ones

43

# Other behaviors

44

## Generic finite state machine (gen_fsm)

- A finite state machine is defined as a set of relations of the form: State(S) × Event(E) → Actions(A) × State(S')
  - "If we are in state S and event E occurs, then we should perform the actions A and make a transition to state S' "
  - Many applications can be modeled as FSMs
- Starting a gen_fsm
- Notifying about events (for actions)
- Stopping a gen_fsm
  - If the gen_fsm is part of a supervision tree, no stop function is needed.
  - If the gen_fsm is standalone, a stop function may be useful

45

## Generic event handler (gen_event)

- An event is a message that is sent when a specific condition occurs
  - For example, an error, an alarm, or information to be logged
- The event manager installs zero or more event handlers
  - When the event manager is notified about an event, all the event handlers will process it
- Starting and stopping an event manager
  - As before, if the gen_event is part of a supervision tree, no stop function is needed

46

# Conclusions

47

## Conclusions

- The Erlang/OTP platform combines the Erlang language with generic OTP libraries for building resilient highly concurrent and distributed systems (www.erlang.org)
  - Erlang supports "processes" (active agents) that share no state and communicate through asynchronous messages
    - Processes are defined using pure functional programming and support dynamic code updating
    - Failure detection is part of the message communication (process linking)
  - OTP supports resilient releases and applications using behaviors, supervisor trees, and testing
    - A behavior is a generic concurrency pattern (server, FSM, event handling)
    - A supervisor tree manages how failure handling is done
    - Testing tools for testing concurrent and distributed applications including fault injection
- Erlang/OTP is being used successfully for many industrial applications, and Erlang ideas are being incorporated into other programming languages and systems
  - Mainstream languages are extended to message passing and agents, and Erlang itself is evolving to resemble mainstream languages (Elixir provides a Java-like syntax with an Erlang semantics)
  - The Erlang Ecosystem Foundation was founded recently to support Erlang and Elixir

48

# Bibliography

49

## Bibliography


- Joe Armstrong, Concurrency Oriented Programming in Erlang, Talk slides, Nov. 2002.
- Joe Armstrong, Making Reliable Distributed Systems in the Presence of Software Errors, Ph.D. dissertation, KTH, Dec. 2003.
- Joe Armstrong, Programming Erlang: Software for a Concurrent World, The Pragmatic Bookshelf, 2007.
- Staffan Blau and Jan Rooth. AXD 301–A New Generation ATM Switching System, Ericsson Review No. 1, 1998.
- Francesco Cesarini and Steve Vinoski. Designing for Scalability with Erlang/OTP, O'Reilly, 2016.
- Ericsson AB. Erlang (Condensed) , Talk slides.
- Ericsson AB. OTP Design Principles.
- Ericsson AB. Erlang/OTP System Documentation Version 10.7, March 2020.
- Frederic Trottier-Hebert. Learn You Some Erlang For Great Good.
- Ulf Wiger. Four-fold Increase in Productivity and Quality, March 2001.


50