LINFO1131
Concurrent programming concepts

Lecture 1: Introduction and refresher

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview

- Course overview
  - Course organization
  - Course content
- Refresher of LINFO1104
  - Functional programming
  - Recursion and invariant programming
  - Higher-order programming
  - Symbolic programming
  - Data abstraction
  - Deterministic dataflow
  - Nondeterminism

2

Course organization

- Organization
  - **Weekly lectures**: presence strongly recommended
  - **Weekly labs**: presence strongly recommended; 1 point bonus if present during all labs
  - **Moodle**: all slides and announcements will be here
- Grading
  - **Midterm**: optional; around week 7; corresponds to 5 points on exam
  - **Project**: mandatory; groups of 2 students; last third of quadrimester; must be done during quadrimester
  - **Exam**: 5 points corresponds to midterm (max of both), 10 points for rest of course

3

Software

- Oz language:
  - Mozart 2 system
  - www.mozart2.org
- Erlang language:
  - Erlang/OTP 25 system
  - www.erlang.org

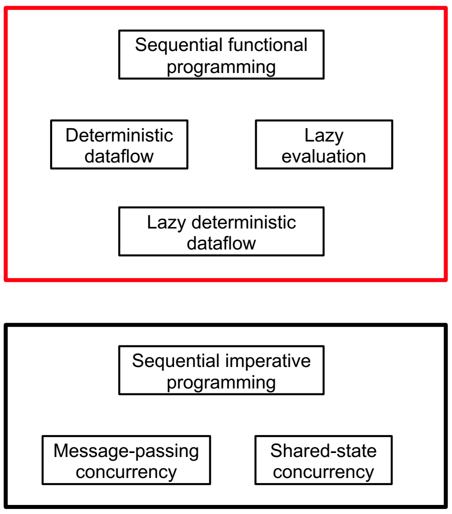
4

Three main themes

- Understanding concurrent programming
  - This is very hard in general
  - We will see how it can be made easy
- Understanding declarative programming
  - This is the easiest and best way to program
  - We will see how to use it as much as possible
- Understanding nondeclarative programming
  - This is quite hard but it cannot always be avoided
  - We will see how to make it reasonably easy

5

Course themes



Sequential functional programming
Deterministic dataflow
Lazy evaluation
Lazy deterministic dataflow
Sequential imperative programming
Message-passing concurrency
Shared-state concurrency

Declarative programming

Functional programming
Higher-order programming
Lazy and concurrent programming
Semantics of declarative concurrency
Advanced declarative algorithm design

Nondeclarative programming

Limitations of declarative programming
Minimizing nondeterminism
Multi-agent programming
Fault tolerance and supervisors, Erlang
Locks, monitors, and transactions

6

Approximate course schedule

- S1: Introduction and refresher

Advanced declarative programming

- S2-S3: Lazy evaluation and lazy deterministic dataflow
- S3: Declarative concurrency
- S4-S5: Advanced declarative algorithm design
- S5: Limits of declarative programming

SMART week

- S7: Midterm

Advanced nondeclarative programming

- S6: Data abstraction
- S8: Multi-agent programming
- S9-S10: Robust multi-agent programming in Erlang
- S11: Shared-state concurrency, locks and tuple spaces
- S12: Monitors
- S13: Transactions

- S14: Course review

7

Refresher of previous course (LINFO1104)

8

Refresher

- In the rest of today's lecture, we recapitulate the main concepts needed for LINFO1131
- These concepts are taught in the previous course LINFO1104
  - If anything is not clear, please look it up!
  - Starting with next week's lecture, I will assume that all of this is *perfectly understood*

9

Functional programming

10

Functional programming

- It is the foundation of all programming
  - Higher-order programming is the foundation of all data abstraction
  - It is the best paradigm for testing, maintenance, and proving correctness
  - It is more and more being used, e.g., cloud analytics based on MapReduce
- In this course we will push it as far as we can
  - For concurrency: deterministic dataflow
  - For efficiency: advanced algorithm design

11

Invariant programming

12

Recursive functions

- In other words, loops!
  - Tail recursion = while loop
- Factorial example

  Invariant: n! = i! x a

  where: n is a constant

  i decreases

  a increases
  - "Principle of communicating vases"
- We give code in C and Oz…

13

Factorial example

- C code:

```
int fact(int n) {
    int a=1;
    int i=n;
    while (i>0) {
        a=i*a;
        i=i-1;
    }
    return a;
}
x=fact(10);
```

- Oz code:

```
fun {Fact A I}
   if (I>0) then
      {Fact I*A I-1}
   else A end
end
X={Fact 1 10}
```

14

Higher-order programming

15

Higher-order programming

- A function is a value in the language
- Key concept for defining data abstractions
- Example: (function as output)

```
fun {AddN N}
   fun {$ X} X+N end
end
Add5={AddN 5}
Add10={AddN 10}
```

- What is the order of AddN?

16

Closures

- Synonyms:
  - Closure ("fermeture")
  - Lexically scoped closure
  - Procedure value
- Closure in memory is code+environment:

```
a1 = (proc {$ X R} R=X+N end, {N→n}),
a2 = (proc ($ X R} R=X+N end, {N→m}),
n=5,
m=10
```

17

Map function

- Example with function as argument:

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

- This function is tail-recursive! Why?

18

More abilities of higher-order

- Higher-order programming can do many things:
  - Genericity
  - Instantiation
  - Function composition
  - Abstracting an accumulator
  - Encapsulation
  - Delayed execution

19

Symbolic programming

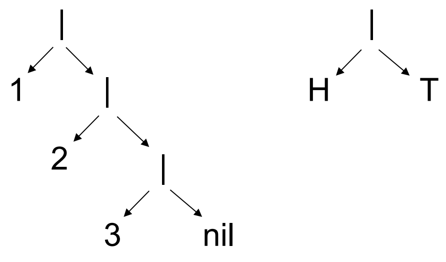
20

List data structure

- A list is a recursive data structure
  <L> ::= nil | <E> '|' <L>
  - Recursive list function follows data definition
- Lists are a ubiquitous data structure
  - Languages give them syntactic supporrt
  - a|b|c|nil, [a b c], '|'(1:a 2:'|'(1:b 2:'|'(1:c 2:nil)

21

Pattern matching



- **case** L **of** H|T **then** … **end**
  - The case instruction matches a pattern, which is a shape of the matched structure
  - Patterns can match or fail to match

22

Fold operation

- The fold operation abstracts an accumulator
  - It uses higher-order programming to encapsulate an accumulator inside its definition
- Given a list $L=[a_0\ a_1\ \ldots\ a_{n-1}]$
  - Define the following function:
    - $s = (\cdots((u\ f\ a_0)\ f\ a_1) \cdot\cdot\cdot)\ f\ a_{n-1})$
    - s = fold(l f u)
- What is the Oz definition of fold?

23

Fold definition

- Recursive function with accumulator:

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

- How can we use FoldL to compute the sum of elements of a list?
  - Is FoldL a tail-recursive function?

24

Data abstraction

25

Data abstraction

- A data abstraction encapsulates part of a program so that it can only be accessed through an interface
  - The interface can only be used through predefined rules
- Advantages of data abstraction
  - Guarantee that the abstraction will work
  - Reduction of complexity
  - Developing large programs with teams

26

Defining a data abstraction

- Data abstractions are defined with two concepts
  - Lexical scoping
  - Higher-order programming
- There are two main kinds of data abstractions
  - Objects
  - Abstract data types

27

A stack as abstract data type

- We define a stack as an ADT: (this is a semantic definition)

```
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)

28

A stack as an object

- We define a stack as an object: (this is a semantic definition)

```
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
```

- How does this work?
- 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

29

Special syntax for objects

- Most widely used languages provide a special syntax for objects
  - The semantics is the same
- This syntax guarantees that a programmer defines the object in the right way!
  - It also makes the program more readable
- Here is the Oz syntax for the stack object defined in the previous slide

```
class Stack
    attr c
    meth init
         c:=nil
    end
    meth push(X)
         c:=X|@c
    end
    meth pop(X)
         S=@c in X:=S.1 c:=S.2
    end
    meth isEmpty(B)
         B=(@c==nil)
    end
end

S={New Stack init}
```

30

Deterministic dataflow

31

Deterministic dataflow

- Concurrency = multiple activities executing simultaneously
- We provide a language operation to create a new concurrent activity:

  thread <s> **end**

  - Instruction <s> is executed independently of other activities

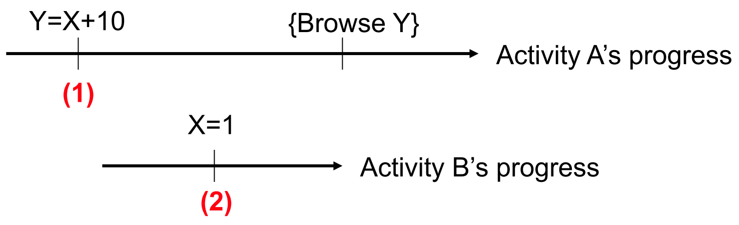
32

Dataflow concurrency

- Functional programming with concurrency
  - Unbound variables and threads
- Threads communicate with each other through shared variables
  - One thread binds the variable and one thread reads the variable
  - **thread** X=1 **end**
    thread Y=X+10 {Browse Y} **end**
  - This program always displays 11! It does not matter in which order the threads execute. Why not?

33

Always the same result!



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

34

Concurrent pipeline

- A *stream* is a list that ends in an unbound variable, which can be used as a channel
- An *agent* is a concurrent activity that reads and writes streams
- All list functions can be used as agents
  - Because list functions are tail-recursive, the agent will have constant stack size. Why are list functions tail-recursive?
  - All functional programming techniques can be used in deterministic dataflow
- Producer:
```
fun {Prod N}
     N|{Prod N+1}
end
```
- Consumer:
```
fun {Cons S}
     case S of H|T then
        {Browse H}
        {Cons T}
     end
end
```
- Pipeline:
```
thread S={Prod 1} end
thread {Cons S} end
```

35

Nondeterminism and the semantics of concurrency

36

Semantics of concurrency

- Each thread executes as a sequence of steps:
  T_1: $e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow \cdots$
  T_2: $e'_0 \rightarrow e'_1 \rightarrow e'_2 \rightarrow \cdots \rightarrow \cdots$
- All threads are executed on one processor
  - The processor is sequential; it executes one instruction sequence (we assume a single core!)
  - Thread executions are *interleaved* on the processor
  - The *scheduler* chooses which thread to execute
  - The scheduler must be *fair* ; for efficiency and practicality, real systems also add *time slices* and *priorities*

37

Nondeterminism

- A program is *nondeterministic* when it executes an operation that is chosen external to the program
  - Scheduler choices are an example of nondeterminism
- Nondeterminism is inherent to any concurrent program
  - Threads are *independent* by definition, so the scheduler must make its choice outside of the control of the programmer
- Nondeterminism is inherent to any program that interacts with the real world
  - In the general case, real world events can happen at any time and must be handled when they occur
  - To interact with the real world, the program must be concurrent

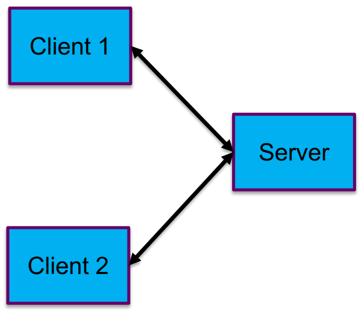
38

Client/server example

- A client/server is a typical example of a program that interacts with the real world and therefore makes choices external to the program
- The nondeterminism is a direct consequence of the client/server specification:
  - When a client makes a request to the server, the server must respond in a timely fashion that depends only on the network travel time and computation time of the request
  - Why is this?

39

Client/server application



- Specification:
  - When a client makes a request to the server, the server must respond in a timely fashion that depends only on the network travel time and computation time of the request
- Therefore the order of the requests cannot be determined in advance because it depends on precise timing of messages and computations
  - The order of message arrival at the server is determined external to the program!
- The whole client/server application is therefore nondeterministic, even if all the other code is purely declarative

40

40

Living with nondeterminism

- A nondeterministic program is *much harder* to develop and debug than a deterministic program
  - The program must work correctly for all possible nondeterministic choices
  - In general, there are very many such choices and they can happen at any time during the execution
    - Testing is very hard: we need to simulate all these choices!
- So what can we do?
  - This is one of the main themes of the course!
  - How to live with nondeterminism

41

How to live with nondeterminism

- A main theme of the course
- Two solutions:
  - **Deterministic dataflow**: this paradigm solves the problem for us. Even though the scheduler is doing nondeterministic choices, the results of the program are always the same. The nondeterminism is hidden because of the strong properties of functional programming.
  - **Nondeclarative programs**: the programmer has to solve the problem. The way to do this is by defining the right abstractions. We will see this later!

42

Conclusions

43

Conclusions

- LINFO1131 continues the story of LINFO1104
  - Concurrent programming
  - Declarative programming
  - Nondeclarative programming
- In today's lecture we explained the goals and organization of LINFO1131
- We recalled the most important concepts of LINFO1104
  - Please refer back to that course if anything is not clear
- Next week's lecture will be on lazy evaluation

44