LINFO1131
Concurrent programming concepts

Lectures 2 and 3: Lazy evaluation and declarative programming

Sep. 28 & Oct. 5, 2022

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

1

Overview

- Introduction to lazy evaluation
  - Semantics based on dataflow
- Lazy streams
  - Three kinds of producer-consumer
  - Infinite lists
  - Hamming problem
- Lazy suspensions
  - Graphical representation of lazy evaluation
- Lazy deterministic dataflow
  - Bounded buffer
- Lazy quicksort
  - Inventing an incremental algorithm
- What is declarative programming?
  - Partial termination
  - Equivalent stores
  - Definition of declarative programming
  - Failure confinement
- Table of declarative paradigms
- Conclusions

2

Introduction to lazy evaluation

3

Introduction to lazy evaluation

- A lazy program is a functional program that executes in "by-need" fashion
  - Nothing is computed until it is "needed"
- Here is a simple example:

```
fun lazy {LazyAdd X Y}
     X+Y
end
S={LazyAdd 10 20}
{Browse S}
```

- Nothing is executed until S is needed:

```
% Displaying the addition S+100 needs S:
{Browse S+100}
```

4

Semantics of lazy evaluation

5

Semantics of LazyAdd

- How does LazyAdd work?
  - Semantics of a program is defined by translation into kernel language
  - We will define what "needing a value" means
- We translate into kernel language:

```
proc {LazyAdd X Y R}
    thread
        {WaitNeeded R} R=X+Y
    end
end
```

- The {WaitNeeded R} waits until another thread needs R to continue
  - More precisely, it waits until another thread does {Wait R}
  - This is part of dataflow execution…

6

Dataflow semantics

- To understand WaitNeeded, we first recall how dataflow execution works
- Given any expression: S=X+Y
- This is translated as:

```
local V in
    {Wait X}
    {Wait Y}
    {PrimitiveAdd X Y V}
    {Bind S V}
end
```

- This gives a dataflow execution:
  - {Wait X} suspends until X is bound
  - {Bind X V} binds X to V
- Programmer-accessible operations are defined using Wait, Bind, and a primitive operation:
  - Arithmetic, boolean expressions
  - Case statements
  - Any operation with an input
    - Function call {F X} where F must be bound to a function value
    - Dot operation R.name where R must be bound to a record

{WaitNeeded X} suspends until another thread does {Wait X}

7

Another example

- We use WaitNeeded directly:

```
declare X in
{WaitNeeded X}
X=100
```

- This displays an unbound variable:
  {Browse X}
- This displays 100 twice (!):
  {Browse X+0}

8

General translation scheme

- Given any lazy function:
```
fun lazy {F X1 … Xn}
    <expr>
end
```
- This is translated into:
```
proc {F X1 … Xn R}
    thread
        {WaitNeeded R} R=<expr>
    end
end
```
- This translation gives the semantics, not the implementation!
  - A compiler is free to optimize it while respecting the semantics

9

Three kinds of producer-consumer

10

Producer-consumer code

- We give the code of a simple producer-consumer
  - We will show three different ways to run the same code
  - All three ways are declarative and end up with the same result
- Technically we are just taking advantage of the Church-Rosser theorem
  - All reduction orders of a lambda expression give the same result
  - Also called confluence

```
fun {Prod L H}
   {Delay 1000} % Wait 1000 ms
   if L>H then nil
   else L|{Prod L+1 H}
   end
end

fun {Cons S Acc}
   case S of H|T then
      Acc+H|{Cons T Acc+H}
   [] nil then nil
   end
end
```

11

Sequential execution

- We generate 10 elements
  - Nothing is displayed until after 10 seconds
  - This is a batch execution

```
declare S1 S2 in
{Browse S1}
{Browse S2}
S1={Prod 1 10}
S2={Cons S1 0}
```

12

Concurrent execution

- We execute both calls in their own threads
  - This is running deterministic dataflow (eager)
  - What is the difference with the sequential version?
    - This is an incremental execution

```
declare S1 S2 in
{Browse S1}
{Browse S2}
thread S1={Prod 1 10} end
thread S2={Cons S1 0} end
```

13

Lazy execution

```
fun lazy {Prod L H}
   {Delay 1000}
   if L>H then nil
   else L|{Prod L+1 H}
   end
end

fun lazy {Cons S Acc}
   case S of H|T then
      Acc+H|{Cons T Acc+H}
   [] nil then nil
   end
end
```

- We annotate both functions as “lazy”
- We execute it:

```
declare S1 S2 in
{Browse S1}
{Browse S2}
S1={Prod 1 10}
S2={Cons S1 0}
```

- What is going on?
  - Why is nothing computed?
  - How do we run this?
  - {Browse S2.2.1} displays the second element of S2, which will activate its computation

14

Eager versus lazy streams

- One way to understand the difference between eager and lazy is to see which agent is driving the execution
- In an eager stream, it is the producer that determines when elements are sent
  - Termination is decided by the producer
- In a lazy stream, it is the consumer that determines when elements are sent
  - Termination is decided by the consumer

15

Infinite lists

16

Infinite lists

- With lazy evaluation we can compute with infinite loops
  - We can write programs with infinite lists
  - It works because the execution will only generate the elements that are needed
- An infinite list of integers starting with N:
  fun lazy {Ints N} N|{Ints N+1} **end**
- Calling {Ints 1} displays an unbound variable:
  L={Ints 1} {Browse L}
- We can force a computation by examining the list L:
  {Browse L.1}
  {Browse L.2.1}

17

Semantics of infinite lists

- We can see how infinite lists work by translating to kernel language:

```
proc {Ints N R}
    thread
        {WaitNeeded R} R=N|{Ints N+1}
    end
end
```

- When we need R by doing {Browse R.1}, this causes R to be bound to N|{Ints N+1}
  - This causes one element of R to be computed
  - The recursive call will immediately suspend again

18

Forcing a computation

- We can force the evaluation of N elements of a list by traversing the list:

```
proc {Touch L N}
   if N==0 then skip
   else {Touch L.2 N-1} end
end
```

- This strange procedure does nothing by itself, yet it forces the work to be done:

```
{Touch L 10}
{Touch L 20}
```


Hamming problem

Hamming problem

- Richard Hamming (1915-1998) was an engineer and mathematician who worked at Bell Labs and invented many useful things
  - Hamming codes, Hamming window, Hamming distance, etc.
  - *The Art of Doing Science and Engineering: Learning to Learn*, by Richard Hamming, 1997. This book is highly recommended!
- Today we will investigate the Hamming problem, a simple problem in number sequences
  - It is a dynamic problem where we do not know in advance how much needs to be computed → perfect for lazy evaluation!
  - We will use lazy evaluation to design a simple and efficient solution to this problem

21

Hamming problem

- Problem statement:
  - Given the set of numbers of the form $2^a\ 3^b\ 5^c$ with integers a,b,c ≥ 0
  - It is asked to compute these numbers in increasing order: 1 | 2 | 3 | …
- We do not know in advance how many numbers of this sequence will be needed
  - The program should let us compute them incrementally until we are satisfied
  - The program should be efficient in time and memory!

22

Algorithm idea

H = 1 | 2 | 3 | X |…

2H = 2 | **4** | 6 | …

3H = 3 | **6** | 9 | …

5H = **5** | 10 | 15 | …

$\Rightarrow$ X = min(4,6,5) = 4

- Idea: The next number X is 2 times, 3 times, or 5 times one of the previous numbers in the sequence
- We need to keep three sequences derived from H, namely 2H, 3H and 5H, and take the least number not yet used
- Numbers 2 and 3 are already taken
- Next number is either 4, 6, or 5
- We take the minimum of these three: the next number is 4

- We can program this with lazy lists

23

Hamming program operations

- The algorithm needs two operations
  - Multiply list elements by an integer
  - Merge two ordered lists
- L2={Times L1 N}
  - Each element of L2 is N times the element of L1
- L={Merge L1 L2}
  - Assume L1 and L2 are in increasing order
  - L contains elements of L1 and L2 in increasing order

24

Hamming program

```
fun lazy {Times S N}
    case S of H|T then
        N*H|{Times T N}
    end
end

fun lazy {Merge S1 S2}
    case S1|S2 of (H1|T1)|(H2|T2) then
        if H1<H2 then H1|{Merge T1 S2}
        elseif H1>H2 then H2|{Merge S1 T2}
        else /* H1==H2 */ H1|{Merge T1 T2}
        end
    end
end
```

- Main expression:

```
H=1|{Merge
        {Times H 2}
        {Merge {Times H 3}
                {Times H 5}}}
{Browse H}
```

25

Lazy suspensions

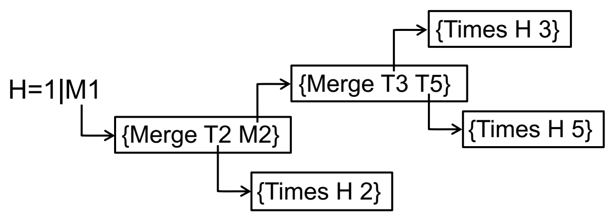
26

Lazy suspensions

- We defined lazy evaluation using threads and WaitNeeded
  - This is correct but it does not show the execution
- Let us show the execution of a lazy program with a graphical approach
- We introduce the concept of lazy suspension:
  Executing: L2={Times L1 3}
  Creates a suspension: L2 → $\boxed{\text{\{Times L1 3\}}}$
  "A thread is suspended on L2 that contains the body of {Times L1 3}"

27

Execution of Hamming program



- Running the program creates five lazy suspensions
  - The lazy suspension {Merge T2 M2} waits on M1
  - Executing M1.1 activates the lazy suspension {Merge T2 M2}, which executes the body of {Merge T2 M2}, which then activates {Merge T3 T5} and {Times H 2}, and so forth!
    - All five lazy suspensions are activated and five new ones are created
    - At the end, M1.1 is bound to 2|M1' with the new variable M1'

28

First activation: {Merge T2 M2}

- Request the second element of H:
  {Browse M1.1}

- This activates {Merge T2 M2}:
  - The body is executed:
    case T2|M2 **of** (H1|T2') | (H2|M2') **then**
        …
    end

  - The **case** needs the first elements of T2 and M2
    - This activates {Times H 2} and {Merge T3 T5}
    - The **case** waits patiently until T2 and M2 are bound

29

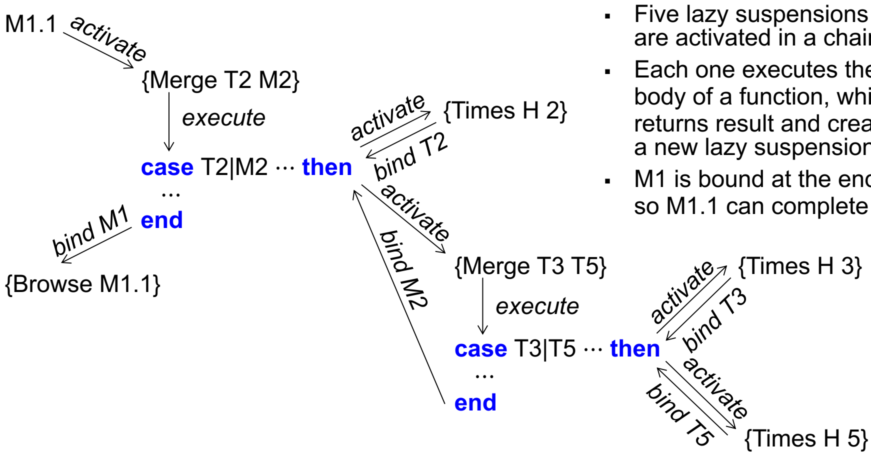
Next activations

- {Times H 2} and {Merge T3 T5} are activated
  - The body of {Times H 2} is executed
    - This binds T2=2|T2' and creates a new lazy suspension on T2': {Times M1 2}
  - The body of {Merge T3 T5} is executed
    - This activates {Times H 3} and {Times H 5}
    - After executing these two functions, this binds M2=3|M2' and creates a new lazy suspension on M2': {Merge T3' T5}
- Now the **case** in {Merge T2 M2}, which was waiting patiently, can be executed:
    - It returns 2|M1' with M1'={Merge T2' M2} and creates a new lazy suspension on M1'

30

Overall execution flow

- Doing M1.1 starts it all
- Five lazy suspensions are activated in a chain
- Each one executes the body of a function, which returns result and creates a new lazy suspension
- M1 is bound at the end so M1.1 can complete



31

Lazy deterministic dataflow

32

Five functional paradigms

- So far we have shown four paradigms of functional programming:
  - Sequential functional programming
  - Sequential functional programming with single assignment
    - Allows data structures with "holes", e.g., list functions are tail-recursive
  - Deterministic dataflow
    - Adds threads and dataflow synchronization
    - Allows concurrent programming with streams
  - Lazy evaluation
    - Adds by-need evaluation (with WaitNeeded), where functions are executed only when their results are needed
    - Allows programming with infinite lists
- There is a fifth paradigm:
  - Lazy deterministic dataflow
    - Adds both threads and lazy functions

33

Lazy deterministic dataflow

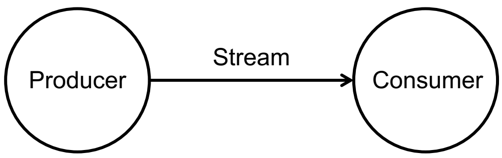
- Lazy deterministic dataflow is the most powerful declarative paradigm:
  - It has the strong properties of functional programming: confluence and higher-order
  - It has concurrency: independent activities which can get out of step with each other
  - It has lazy evaluation: by-need computations which are only done when their result is needed
- What can we do with all this power?
  - We give one example of a program that can be written in lazy deterministic dataflow, but not in any weaker paradigm

34

Bounded buffer

35

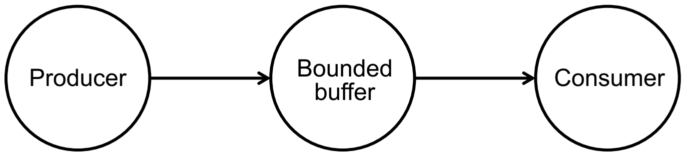
Bounded buffer (1)



- A producer-consumer pipeline has performance problems
  - Variations in producer and consumer speeds can cause the system to perform poorly
  - When a producer creates elements too quickly, the consumer cannot use the elements so the producer idles
  - When a consumer needs more elements, the producer may not be able to produce them so the consumer idles

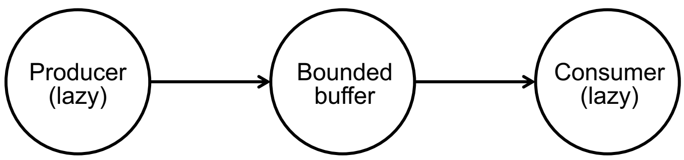
36

Bounded buffer (2)



- Inserting a bounded buffer can solve these problems
  - When the producer creates elements too quickly to be consumed, they are stored in the bounded buffer
  - When the consumer needs more elements than can be produced, they are taken from the bounded buffer
  - This improves performance by smoothing out fluctuations in producer and consumer speeds

Bounded buffer (3)



- A bounded buffer fits in between a lazy producer and a lazy consumer
- The code of the producer and consumer is unchanged
  - To the producer, the bounded buffer looks like a consumer
  - To the consumer, the bounded buffer looks like a producer
- The bounded buffer "consumes" elements even when the consumer does not ask for them, and "produces" elements even when the producer does not make them

Defining the bounded buffer

- Assume we have a producer-consumer pipeline:
  thread S={Producer …} **end**
  thread {Consumer S} **end**

- The bounded buffer is inserted in between:
  thread S1={Producer …} **end**
  thread {BoundedBuffer S1 S2 10} **end**
  thread {Consumer S2} **end**

- We define the bounded buffer step-by-step
  - We define the procedure {BoundedBuffer S1 S2 N} where S1 is the input stream, S2 is the output stream, and N is the buffer size
  - We build the procedure in four steps, to make it easier to understand

39

First step: pass elements

- The buffer outputs the same elements as it inputs:

```
proc {BoundedBuffer S1 S2 N}
   fun lazy {Loop S1}
      case S1 of H1|T1 then H1|{Loop T1} end
   end
in
   S2={Loop S1}
end
```

40

Second step: startup

- The buffer asks for N elements on startup:

```
proc {BoundedBuffer S1 S2 N}
   fun lazy {Loop S1}
      case S1 of H1|T1 then H1|{Loop T1} end
   end
   End
in
   End={List.drop S1 N} % Asking must not be lazy!
   S2={Loop S1}
end
```

- {List.drop L N} is a library function that removes the first N elements from a list L

41

Third step: staying full

- Whenever the consumer gets an element, the buffer asks for another element from the producer:

```
proc {BoundedBuffer S1 S2 N}
   fun lazy {Loop S1 End}
      case S1 of H1|T1 then H1|{Loop T1 End.2} end
   end
   End
in
   End={List.drop S1 N}
   S2={Loop S1 End}
end
```

42

Fourth step: no blocking

- To avoid blocking the buffer's main loop, both asks must be done in their own threads:

```
proc {BoundedBuffer S1 S2 N}
   fun lazy {Loop S1 End}
      case S1 of H1|T1 then
         H1|{Loop T1 thread End.2 end}
      end
   end
   End
in
   thread End={List.drop S1 N} end
   S2={Loop S1 End}
end
```

In declarative programming, threads are your friends! They are efficient. They can be added at will without adding bugs. They remove blocking and make the program more incremental.

All list functions, including List.drop, work correctly when used concurrently

43

Example execution

- We create a pipeline with producer, bounded buffer, and consumer:

```
declare S1 S2 S3 in
{Browse S1}
{Browse S2}
{Browse S3}
S1={Prod 1 10}
{BoundedBuffer S1 S2 3}
S3={Cons S2 0}
```

- Note that the producer immediately produces 3 elements, which are stored in the buffer
- When we consume one element, the buffer asks the producer for one element
  - The buffer tries to stay full
- The buffer is eager until it is full, and then it becomes lazy

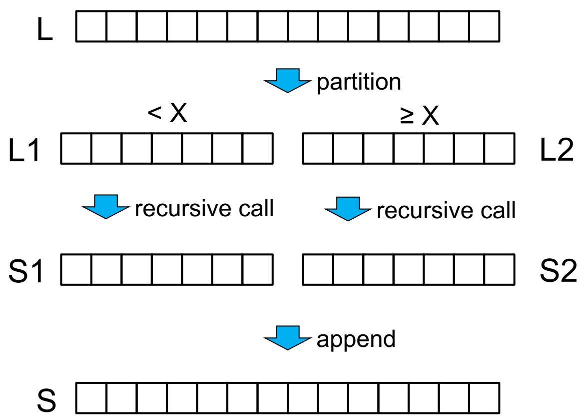
44

Lazy quicksort

Lazy quicksort

- Lazy evaluation can make some algorithms incremental, which can enormously improve their efficiency
  - We show this with the quicksort algorithm
- Standard quicksort has an average time complexity of O(n log n) to sort n elements
- Lazy quicksort has a time complexity of O(n + k log k) to compute the k smallest elements out of n elements
  - This is a very good bound!
  - Furthermore, the value of k does not need to be known in advance. Elements can be computed incrementally until some condition is satisfied.
  - To see how clever this is, try inventing the algorithm from scratch!

Quicksort algorithm



- Pick a random element of L, the "pivot" X
- Partition into two sublists
- Recursively sort the sublists
- Append the results

Quicksort example (on board)

- L = [7 3 2 8 6 4 1 9]
- Pivot = 7 (first element of L)
- L1 = [3 2 6 4 1], L2 = [7 8 9]
- …
- S1 = [1 2 3 4 6], S2 = [7 8 9]
- S = [1 2 3 4 6 7 8 9]

Partition procedure

```
proc {Partition L X L1 L2}
   case L of H|T then
      if H<X then M1 in
         L1=H|M1 {Partition T X M1 L2}
      else /* H≥X */ M2 in
         L2=H|M2 {Partition T X L1 M2}
      end
   [] nil then L1=nil L2=nil
   end
end
```

49

Append and quicksort

```
fun {Append L1 L2}
   case L1 of H|T then H|{Append T L2}
   [] nil then L2 end
end
fun {Quicksort L}
   case L of X|M then L1 L2 S1 S2 in
      {Partition L X L1 L2}
      S1={Quicksort L1}
      S2={Quicksort L2}
      {Append S1 S2}
   [] nil then nil
   end
end
```

50

Example eager execution

- Let us try to run this:
  declare S in
  S={Quicksort [4 3 2 5 6 4 3 2]}
  {Browse S}
- What happens?
  - Something is wrong!
- How do we fix this?
  - A general rule when defining recursive functions!

51

Append and quicksort (fixed)

```
fun {Append L1 L2}
   case L1 of H|T then H|{Append T L2}
   [] nil then L2 end
end
fun {Quicksort L}
   case L of X|M then L1 L2 S1 S2 in
      {Partition M X L1 L2}
      S1={Quicksort L1} % L1 is strictly smaller than L
      S2={Quicksort L2} % L2 is strictly smaller than L
      {Append S1 X|S2}
   [] nil then nil
   end
end
```

M is strictly smaller than L

52

Making quicksort lazy

- What has to be made lazy?
  - Quicksort function becomes LQuicksort
  - Append function becomes LAppend
- Partition is not lazy
  - Sorting cannot work unless we look at all the elements of L
  - Partition keeps the same eager definition
  - We create the complete sublists L1 and L2

53

Lazy append and quicksort

```
fun lazy {LAppend L1 L2}
   case L1 of H|T then H|{LAppend T L2}
   [] nil then L2 end
end
fun lazy {LQuicksort L}
   case L of X|M then L1 L2 S1 S2 in
      {Partition M X L1 L2}
      S1={LQuicksort L1}
      S2={LQuicksort L2}
      {LAppend S1 X|S2}
   [] nil then nil
   end
end
```

54

Example lazy executions

- Lazy append:
  declare S **in**
  S={LAppend [1 2 3] [4 5 6]}
  {Browse S}
  - What happens when asking for elements?
- Lazy quicksort:
  declare S **in**
  S={LQuicksort [4 3 2 5 6 4 3 2]}
  {Browse S}
  - What happens when asking for the first element?
  - How much computation is done? What is the time complexity?

55

Execution steps…

```
S={LQuicksort [2 3 4 1]} % Lazy suspension on S
{Browse S.1}
   % S is needed, so execute body of S={LQuicksort [2 3 4 1]}:
   {Partition [3 4 1] 2 L1 L2}
   S1={LQuicksort [1]} % Lazy suspension on S1
   S2={LQuicksort [3 4]} % Lazy suspension on S2
   S={LAppend S1 2|S2} % Lazy suspension on S
      % S still needed, so execute body of LAppend:
      case S1 of H|T then H|{LAppend T 2|S2}
      [] nil then 2|S2 end
         % S1 is needed, so execute body of S1={LQuicksort [1]}
         {Partition nil 1 nil nil}
         S1'={LQuicksort nil} % Lazy suspension on S1'
         S2'={LQuicksort nil} % Lazy suspension on S2'
         S1={LAppend S1' 1|S2'} % Lazy suspension on S1
            % S1 still needed, so execute body of LAppend:
            case S1' of H'|T' then H'|{LAppend T' 1|S2'}
            [] nil then 1|S2' end
               % S1' is needed, so execute body of S1'={LQuicksort nil}:
               case nil of X'|M' then (…)
               [] nil then nil end
               % Now we can do bindings:
               S1'=nil
         S1=1|S2'
   S=1|{LAppend nil 2|S2}
{Browse (1|…).1}
% Displays 1
```

- Follow carefully what is happening
  - When S is needed, it stays needed!
  - We focus on the lazy suspensions
- S → {LQuicksort [2 3 4 1]}
  S is needed, activates:
  - S1 → {LQuicksort [1]}
  - S2 → {LQuicksort [3 4]}
  - S → {LAppend S1 2|S2}
    S still needed, activates:
    S1 is needed, activates:
    - S1'→{LQuicksort nil}
    - S2'→{LQuicksort nil}
    - S1→{LAppend S1' 1|S2}
      S1 still needed, activates:
      S1' is needed, activates:
      - S1'=nil
    - S1=1|S2'
  - S=1|{LAppend nil 2|S2}

56

Complexity of lazy quicksort

Partition [3 4 1]

Partition [1] | Partition [3 4] | n traversal

Partition nil | Partition nil | n/2 traversal
(n/4, n/8, continues to 1)

nil | 1|nil

1|…

1|…

- To compute the smallest element, the number of operations is n + n/2 + n/4 + … + 1 = 2n, so the time complexity is O(n)
- To compute the k smallest elements, a full "mini quicksort" is done as soon as the partitioned list has at least k elements, so the extra time complexity is O(k log k)
- Total time complexity is O(n + k log k)

57

What is declarative programming?

58

Declarative programming

- We have seen five functional paradigms
  - Sequential functional programming
  - Sequential functional programming with single assignment
  - Deterministic dataflow
  - Lazy evaluation
  - Lazy deterministic dataflow
- We claim that they are all declarative
  - What does this mean, exactly?
  - Let us define it starting from the functional programming paradigm
- We show how to classify declarative paradigms according to their concepts and expressive power (Section 4.5.2 in the book)

59

Functional programming

- All functional programs can be encoded as λ expressions
- Church-Rosser theorem:

$$
\begin{array}{ccccccccc}
e_a & \rightarrow & e_2 & \rightarrow & e_3 & \rightarrow & \cdots & \rightarrow & e_b \\
\downarrow & & & & & & & & \downarrow \\
e_2' & & & & & & & & \\
\downarrow & & & & & & & & \vdots \\
e_3' & & & & & & & & \\
\downarrow & & & & & & & & \downarrow \\
\vdots & & & & & & & & \vdots \\
e_c & \rightarrow & e_{c1} & \rightarrow & e_{c2} & \rightarrow & \cdots & \rightarrow & e_d
\end{array}
$$

- If e_a reduces to e_b (in 0 or more steps) and e_a reduces to e_c (in 0 or more steps), then there exists a term e_d such that e_b and e_c can reduce to e_d
- We say the λ calculus is confluent; it has the Church-Rosser property

60

Other functional paradigms?

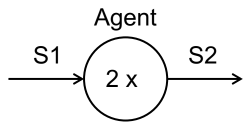
- We see that functional programs are confluent
  - The meaning is clear for the first paradigm, namely sequential functional programming
- But what does it mean for:
  - Concurrency?
  - Streams? (programs never terminate!)
  - Single-assignment variables? (variables can be unbound!)
- We give a precise formal definition of “declarative programming” which covers these concepts
  - Confluence: this handles concurrency (why?)
  - Partial termination
  - Equivalent stores

61

I: Partial termination

62

Partial termination



- Assume we have a concurrent agent with an input stream S1 and an output stream S2
- It could execute as follows:
  - S1=1|_ S2=2|_
  - S1=1|2|_ S2=2|4|_
  - S1=1|2|3|_ S2=2|4|6|_
- How is this functional?
  - The program never terminates and the streams contain unbound variables
- With the right concepts, we can see this as functional execution:
  - If S1 does not change, then S2 reaches a final value
  - We call this “partial termination”
  - We say the program has reached a “resting point”
- What about the unbound variables?

63

II: Equivalent stores

64

Single-assignment variables

- We claim that a functional program that uses single-assignment variables is still functional
  - Let's see how to make this precise
- Consider the following program:
  T_1: **thread** X=foo(Z W) **end**
  T_2: **thread** Y=foo(Z W) **end**
  T_3: **thread** X=Y **end**
  - Assume T_1 and T_2 execute before T_3, then we have the store: σ={*x*=foo(*z w*), *y*=foo(*z w*)}
  - Assume T_1 and T_3 execute before T_2, then we have the store: σ'={*x*=foo(*z w*), *y*=*x*}
- How can we express that stores σ and σ' are the same?

65

A store is a logical formula

- Assume we have these two stores:
  σ={*x*=foo(*z w*), *y*=foo(*z w*)}
  σ'={*x*=foo(*z w*), *y*=*x*}
- The bindings of *x* and *y* are different for σ and σ' but the possible values of *x* and *y* are the same in both stores
  - Let's see how to make this intuition precise
- A store σ corresponds to a relationship between values
  - The store σ tells us that *x* is a record with label foo and arguments *z* and *w*, and that *y* is a record with label foo and arguments *z* and *w*
  - For any values of *x*, *y*, *z*, and *w*, there are two possibilities: either they can be in the store σ or they cannot be in the store σ
  - So the store σ is a logical formula, which can be true or false
  - We write σ as a logical formula: $\sigma \equiv x=\text{foo}(z\ w) \land y=\text{foo}(z\ w)$

66

Interpretation and model

- Definition: An interpretation of a store σ
  - An interpretation of a store σ is an assignment to all symbols in σ
    - For all variables *x* in σ, assign a value **x** to *x*
    - For all record symbols *f* in σ, assign a function **f** to *f* that has the same number of arguments as the record symbol and that returns a value
  - Any interpretation of a store σ is either **true** or **false**
    - A binding $x=f(x_1 \ldots x_n)$ is **true** if the value returned by $\mathbf{f}(\mathbf{x}_1 \ldots \mathbf{x}_n)$ is equal to **x**; otherwise it is **false**
    - A store $\sigma = (x=f(x_1 \ldots x_n) \wedge \ldots \wedge z=f(z_1 \ldots z_n))$ is **true** if all bindings are **true**, otherwise it is **false**

- Definition: A model of a store σ
  - A model of σ is an interpretation in which σ is **true**

67

Equivalent stores

- Now we can define when two stores are equivalent
  - Each store represents a logical formula that can be **true** or **false**
  - Two stores are equivalent when, no matter how we assign values to their symbols, they are either both **true** or both **false**
    - I.e., we cannot find values such that one store is **true** and the other is **false**

- We state this definition using the model concept
  - We introduce the notation α⊨β which means "β is true in all models of α"

- Definition: Two stores σ and σ' are logically equivalent if
  - σ⊨σ' and σ'⊨σ σ' is **true** in all models of σ and σ is **true** in all models of σ'
- Another way to write this is:
  - ⊨ (σ ⇔ σ') (σ ⇔ σ') is a tautology, i.e., it is **true** in all models

68

First-order logic

- To define equivalence of stores, we have introduced a little bit of first-order logic
  - Logical formula (syntax)
    - An expression that denotes a relationship between variables
  - Model (semantics)
    - A set of values and functions in which logical formulas are true or false
- If you want to understand more, you need to study first-order logic!
  - There are programming languages that are based on first-order logic, such as Prolog, constraint programming, and SQL

69

III: Definition of declarative programming

70

Definition of declarative programming

- Now we can define precisely what declarative programming means

A program is declarative if for all possible inputs:
- All executions for those inputs either:
  - do not terminate, or
  - all reach partial termination and give logically equivalent stores

- Remarks:
  - "All executions" means all possible choices of the scheduler
  - We say that a declarative program has "no observable nondeterminism"
  - All five functional paradigms are declarative

71

IV: Failure confinement

72

Fixing a buggy application

- Declarativeness is an extremely powerful property
  - How do we write applications to be as declarative as possible?
    - This is a major theme of the course! “All programs should be declarative except where they interact with the real world.”
  - How do we fix an application that becomes nondeclarative?
    - We can do failure confinement

- Nondeclarative behavior
  - We will see later in the course that applications that interact with the real world can be nondeclarative
    - That kind of nondeclarativeness is unavoidable but can be minimized
  - Right now, let us see what happens when an application has a bug that makes it nondeclarative

73

Bugs are unavoidable

- “It is a truth universally acknowledged, that a program of a certain size must have bugs”
  - With apologies to Jane Austen 😀
- Assume we have the following (simplified!) buggy program:
  ```
  thread X=1 end
  thread Y=2 end
  thread X=Y end
  ```
- This program will always raise an exception
  - Three stores are possible depending on the scheduler choices: $\sigma_1=\{x=1,y=2\}$, $\sigma_2=\{x=1,y=1\}$, $\sigma_3=\{x=2,y=2\}$
  - This is an observable nondeterminism, so it is nondeclarative
- We can fix this by doing failure confinement
  - We will hide the nondeterminism from the rest of the program
  - That way the program becomes declarative again

74

Failure confinement

- The program has three parts that can become inconsistent if there is a bug
  - We use exceptions to protect these parts

```
thread try X1=1 S1=ok catch _ then S1=error end end
thread try Y1=2 S2=ok catch _ then S2=error end end
thread try X1=Y1 S3=ok catch _ then S3=error end end
if S1==error orelse S2==error orelse S3==error then
   X=1  Y=1  /* default result when there is an error */
else
   X=X1  Y=Y1  /* correct result when there is no error */
end
```

75

Table of declarative paradigms

76

Declarative paradigms

	sequential with values	sequential with values and dataflow variables	concurrent with values and dataflow variables
eager execution (strictness)	strict functional programming (e.g., Scheme, ML) (1)&(2)&(3)	declarative model (e.g., Chapter 2, Prolog) (1), (2)&(3)	data–driven concurrent model (e.g., Section 4.1) (1), (2)&(3)
lazy execution	lazy functional programming (e.g., Haskell) (1)&(2), (3)	lazy FP with dataflow variables (1), (2), (3)	demand–driven concurrent model (e.g., Section 4.5.1) (1), (2), (3)

(1): Declare a variable in the store
(2): Specify the function to calculate the variable's value
(3): Evaluate the function and bind the variable

(1)&(2)&(3): Declaring, specifying, and evaluating all coincide
(1)&(2), (3): Declaring and specifying coincide; evaluating is done later
(1), (2)&(3): Declaring is done first; specifying and evaluating are done later and coincide
(1), (2), (3): Declaring, specifying, and evaluating are done separately

77

Conclusions

78

Conclusions

- Lazy evaluation
  - Functions are evaluated only if their results are needed
  - This extends deterministic dataflow with the WaitNeeded operation
  - Programs can use infinite lists and be made more incremental
  - Lazy evaluation can be combined with concurrency
- Declarative programming
  - An application should be declarative except for real-world interaction
  - We need to define precisely what is declarative programming
    - We give a precise definition of declarative programming using the concepts of confluence, partial termination, and logical equivalence
  - Declarativeness is an observational concept: a program can behave declaratively even if it is written in a nondeclarative paradigm
- Next lecture: Advanced declarative algorithm design
  - Declarative algorithms can be as efficient as nondeclarative algorithms

79