# Concurrent programming with functional dataflow

## Collected lecture slides from LINFO1104 and LINFO1131

May 3, 2024

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

https://webperso.info.ucl.ac.be/~pvr/pldc.html
peter.vanroy@uclouvain.be

1

1

# Concurrent programming with functional dataflow

- Functional dataflow is a form of pure functional programming with threads and dataflow synchronization on single-assignment variables
  - It has good properties that make it an excellent choice for concurrent programs
    - We call it "Concurrency for Dummies": threads can be added at will without introducing bugs, all list functions can become efficient concurrent agents with input and output streams by running them in threads, lazy evaluation is available, etc.
  - Its limitation is that it cannot do general (nondeterministic) real-world interaction
    - We define an extension, functional dataflow with ports, which is able to interact with the real world. Ports are a named extension of streams.
    - Realistic applications only need nondeterministic real-world interaction in a few places, so we can use functional dataflow mostly, and functional dataflow with ports for the rest
- This document collects all the course material on functional dataflow in the courses LINFO1104 and LINFO1131, second and third-year university programming courses given at the Louvain Engineering School, UCLouvain
  - This document extends and completes the material on functional dataflow given in Chapter 4 of *Concepts, Techniques, and Models of Computer Programming* (MIT Press)
  - All code examples can be run in the Mozart 2 system, available at www.mozart2.org

2

2

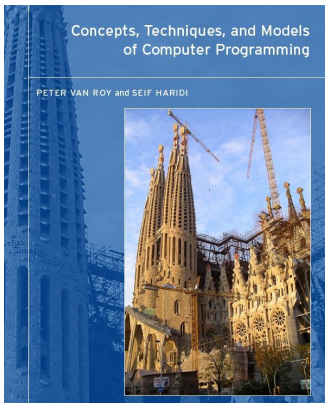
# Table of contents







# Reference book




- Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.
- Chapter 4, “Declarative Concurrency”, covers functional dataflow
- The course slides extend the presentation given in the textbook
  - The textbook gives a formal semantics and situates functional dataflow in the context of general programming with other paradigms

# LINFO1104: Lecture 9
# Functional dataflow





## The world is concurrent

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

# Concurrent programming

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

7

7

# *Concurrency can be (very) hard*

- It introduces many difficulties such as nondeterminism, race conditions, reentrancy, deadlocks, livelocks, fairness, consistency of shared data
  - Java's synchronized objects (monitors) are tough to program with ← LINFO1131
  - 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 (networked systems) makes it even harder
- Adding partial failure makes it even much harder than that

LINFO2345

- 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 functional dataflow, which is a concurrent paradigm that is a form of functional programming

8

8

# Functional dataflow
## (a.k.a. deterministic dataflow)





# Concurrency paradigms

- There are three main paradigms of concurrent programming
- Functional dataflow (the simplest and best)
  - This paradigm is also called deterministic dataflow
  - It supports all the techniques of functional programming
  - That is what we will see today
- What are the two other paradigms?
  - Message-passing concurrency (e.g., Erlang and Scala actors)
    - Activities send messages to each other (like sending letters)
    - This works well and is not too hard
    - Functional dataflow with ports is a refinement that is often better
  - 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 than the two previous paradigms
    - Unfortunately, many current languages still use this paradigm

We will see Erlang later

LINFO1131

# An unbound variable

- Let us explain dataflow by starting with an unbound variable
- An unbound variable is created in memory but not bound to a value
- What happens when you invoke an operation with an unbound variable?

```
local X Y in
    Y=X+1
    {Browse Y}
end
```

- What happens?

11

11

# 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 pauses just before the addition and continues when X is bound (dataflow execution)
  - In constraint programming, the equation "Y=X+1" is added to the set of constraints and execution continues. An amazing way to compute!

LINFO2365 Constraint programming

12

12

# 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?

13

13

# Continuing the execution

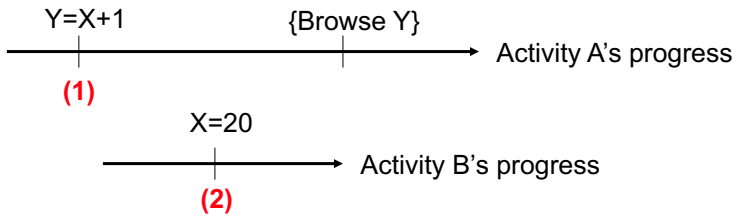
- The waiting instruction:
```
declare X
local Y in
    Y=X+1
    {Browse Y}
end
```
- If someone would bind X, then execution could continue
- But who can do it?

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

14

14

# 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

15

15

# Threads

16

16

# 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 runs independently 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 switching rapidly 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

17

17

# 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!

18

18

## 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

19

19

## 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)

20

20

## 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

21

21

## 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 _ _ ]

22

22

## 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

23

23

## 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]

24

24

# The Browser is a dataflow program

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

25

25

# Streams and agents

26

26

## Streams

- A stream is defined as 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

27

27

## 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!

28

28

# Producer/consumer (1)

- A producer generates a stream of data

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

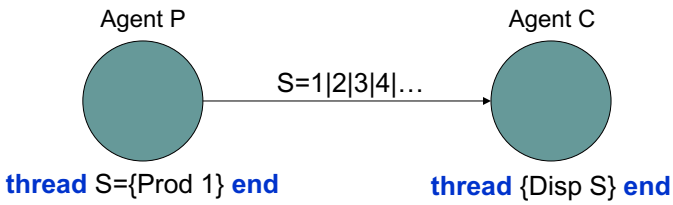
  - The {Delay 1000} slows down execution enough to observe it
- A consumer reads the stream and performs some action (like the Disp procedure)
- A producer/consumer program:

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

29



# 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

30

# 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
```

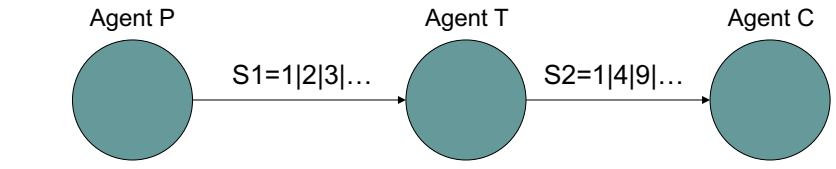
- This program has three agents:

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

31

31

# Pipeline (2)




**thread** S1={Prod 1} **end** **thread** S2={Trans S1} **end** **thread** {Disp S2} **end**

- 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

32

32

# Agents

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

33

33

# Thread semantics

34

34

# Thread semantics (1)

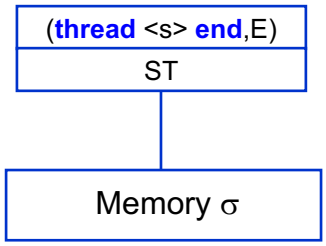
- We extend the abstract machine with threads
- Each thread has one semantic stack
  - The instruction **thread** <s> **end** creates a new stack
  - All stacks share the same memory
- There is one sequence of execution states, and threads take turns executing instructions
  - $(MST_1,\sigma_1) \rightarrow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

35

35

# Thread semantics (2)

A semantic stack that is about to create a thread

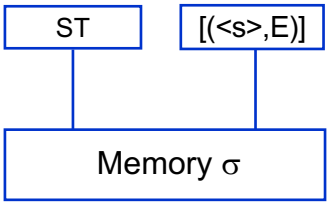



36

36

# Thread semantics (3)

We now have two stacks!




37

37

# Why interleaving semantics?

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

38

38

# Concurrent program execution





# Total order of a sequential program

- A sequential program is a program with one thread
- In a sequential program, execution states are in a total order
  - Total order = there is a defined order between all pairs of states



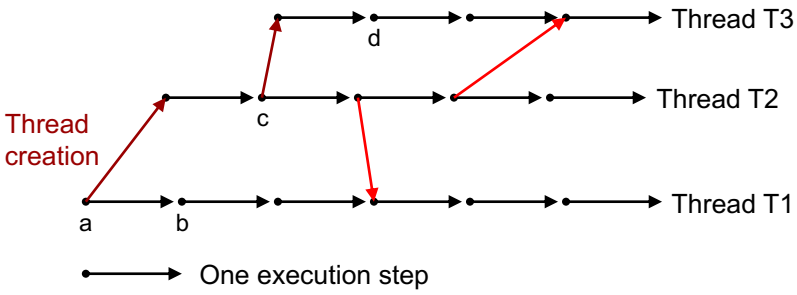
Sequential execution



One execution step

# Partial order of a concurrent program

- A concurrent program is a program with more than one thread
- In a concurrent program, execution states are in a partial order
  - Partial order = not all pairs of execution states have a defined order
  - For example, c < d (c before d) but b and c have no order




41

41

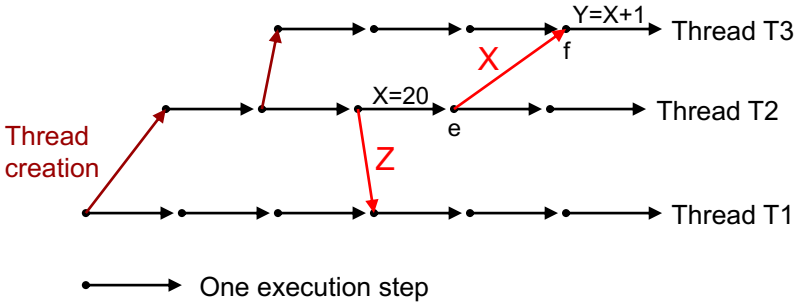
# Partial order of a concurrent program

Wait for the value of a dataflow variable ("Y=X+1")

X

Bind a dataflow variable ("X=20")

- Dataflow synchronization adds order
- For example, e<f because of dataflow

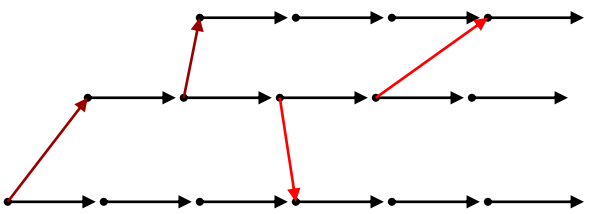



42

42

# The actual execution order

- The processor execution is always one sequence of execution states
- The scheduler chooses the order of these states, which is always compatible with the partial order (choice = nondeterminism)



Partial order of a concurrent program as defined by its threads

*Scheduler **choice** (nondeterminism)*



Total order of actual execution on the processor (one core)[43]

43

# Nondeterminism and the scheduler

44

44

# Nondeterminism and the scheduler

- Nondeterminism is the ability of a system to make decisions (choices) independently of the system's developer
- The scheduler is the part of the system that decides at each moment which thread to execute
  - This decision is an example of nondeterminism
  - Scheduler decisions often vary from one execution to the next; they depend on external conditions such as processor load, memory behavior (caching), network behavior, and timing of external events
- Nondeterminism exists in all concurrent systems
  - It must be so, since the concurrent activities are independent
  - All concurrent programs must manage their nondeterminism!

45

45

# 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

46

46

# 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 also an example of nondeterminism
  - A *choice* made by the system during execution

47

47

# Example of nondeterminism (3)

- What does the following program do?

```
declare X={NewCell 0}
thread X:=1 end
thread X:=1 end
```

- It makes a choice, just like the previous program
  - But in this case, the final results are the same (by accident)
- This is still nondeterminism!
  - The important point is the choice: the running program still sees a difference in the threads' execution order
  - The results may be the same by accident (depending on the computations done), but the choice remains

48

48

# Managing nondeterminism

- Nondeterminism *must always be managed*
  - It should not affect program correctness (this can be very tricky!)
  - The most complicated case is threads and cells used together (like the previous example)
  - Unfortunately, this is exactly how many languages handle concurrency (Java, C++, C#, etc.) ➞ see course LINFO1131
- Functional dataflow has a major advantage
  - The result of a program is **always the same** (*if the program has no error, i.e., raises no exception: errors can change the result*)
  - The nondeterminism of the scheduler does not affect the result
    - There is no observable nondeterminism
    - We call this « Concurrency for Dummies »
    - It is a consequence of Church-Rosser (functional programming)

49

49

# How the scheduler works (1)

- The choice of which thread to execute and for how long is made by the scheduler
- Time slices (on modern systems this is often 10ms)
  - Each thread executes during a short time period called a time slice
  - On multicore processors, some operating systems can allow time slices on different cores, but there is still interleaving semantics
- Thread states (runnable and suspended)
  - 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.

50

50

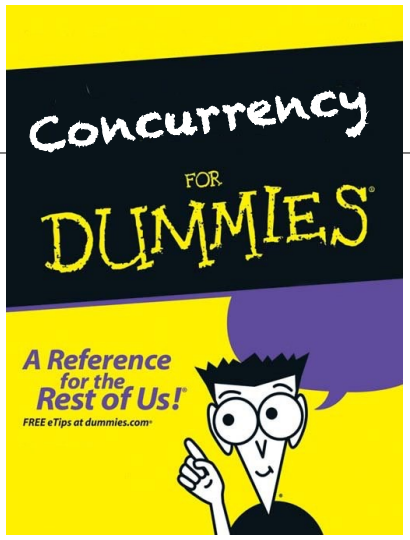
# How the scheduler works (2)

- Fairness
  - A scheduler is fair if every runnable thread will eventually (eventually = in finite time) be executed
- Priority
  - 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
  - Mozart has three priorities, high (≥90%), medium (≥9%), and low (≥1%)
- Reasoning about programs
  - If the scheduler is fair, then it is possible to reason about program execution (since all threads will run to completion)
  - If the scheduler is not fair, then a perfectly correct program may not run correctly
    - Certain threads may starve, i.e., receive 0% of the processor time, so they never execute, and the program just stops

51

51

# "Concurrency for Dummies"




52

52

# "Concurrency for Dummies"

- The multi-agent programs we saw so far are all deterministic
  - Their nondeterminism is not observable (results are 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 changes only the order of computations
  - In functional dataflow, scheduler choices cannot change the result
  - It is possible to add threads at will to a program without changing the result (we call this Concurrency for Dummies)
  - Adding threads can make the program more incremental (by removing unnecessary blocking)
- This only works in functional dataflow (functional programming)!
  - It is a consequence of the Church-Rosser theorem
  - It is not true when using cells and threads together: it does not work in Java, Python, or C++

53

53

# Example (1)

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

54

54

## Example (2)

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

55

55

## 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

56

56

# Example (4)

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

- Compare these two executions: (what do they display?)

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

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

57

57

# 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 Map call would block.

58

58

# 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**

59

59

# 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]

60

60

# 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 cells (= mutable variables)!
  - 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 functional dataflow.

61

61

# 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
```

62

62

# Why does it work? (2)

Translating to kernel language shows how it works

```
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
```

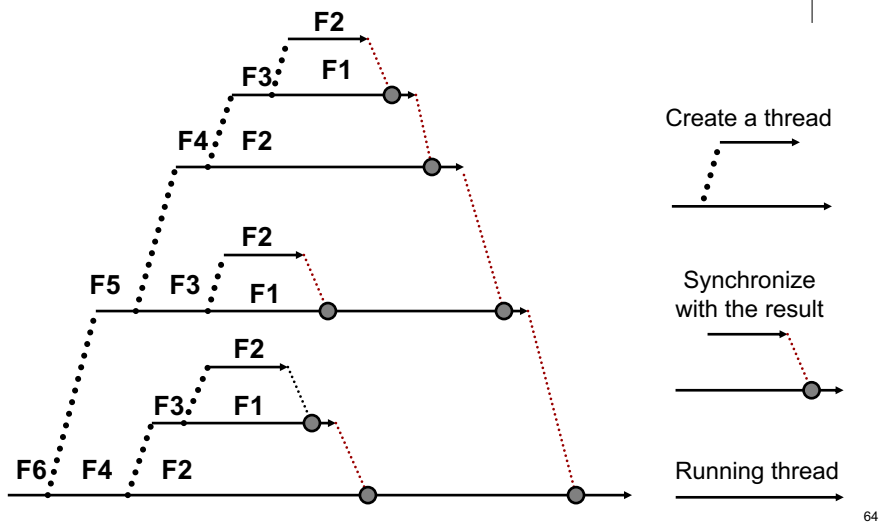
Dataflow dependency

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

63

63

# Execution of {Fib 6}




Create a thread

Synchronize with the result

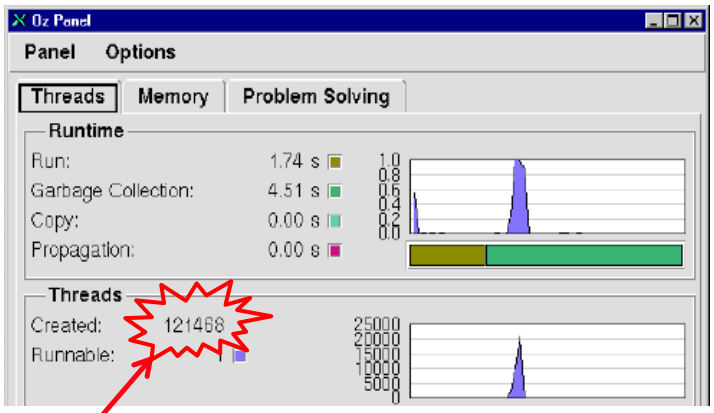
Running thread

64

64

# Observing the execution of Fib

Only in Mozart 1




Total number of threads created since system startup

Oz Compiler Panel (in Oz menu)

65

65

# Counting threads

```
C={NewCell 0}
proc {Inc C}
   {Exchange C X Y} Y=X+1
end

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

This works also in Mozart 2

C:= @C + 1 is not correct!
It is because the scheduler can put the thread to sleep after the X=@C+1 and before the C:=X.

66

66

# Multi-agent programming

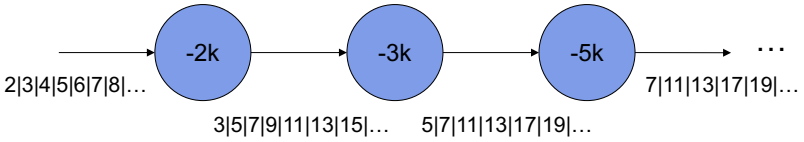




# Multi-agent programming

- 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

# 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

69

69

# 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
```

70

70

# 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)





# 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?)

# Digital logic simulation

73

73

## Digital logic simulation

- The functional 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

74

74

# 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

75

75

# Digital signals as streams

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

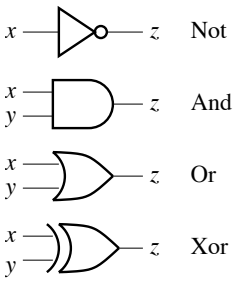
  $S=a_0|a_1|a_2|...|a_i|...$

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

76

76

# Digital logic gates




| x | y | z: Not | And | Or | Xor |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 |

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

77

77

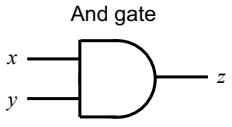
# Digital gates as agents

- 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
```




78

78

# Creating many gates

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

79

79

# 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
```

80

80

## 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}
```

81

81

## Combinational logic

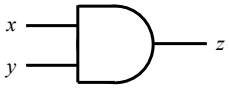
82

82

# 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 $y_i$

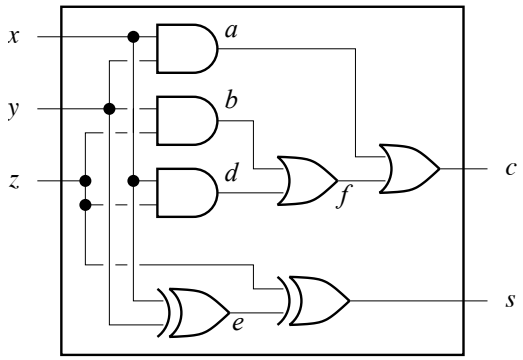



- Therefore, any number of interconnected gates also defines a combinational function
- We define a useful circuit called a full adder

83



# 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

84

# 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
```

85

85

# Sequential logic

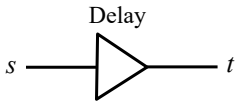
86

86

# 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

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



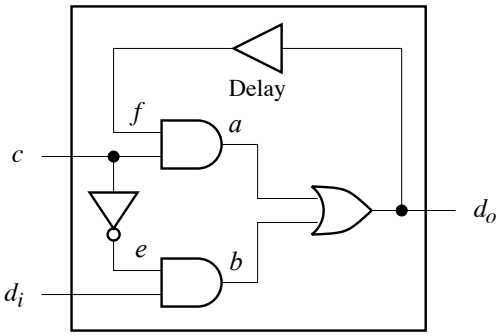

**fun** {DelayG S} 0|S **end**

$b_i=a_{i-1} \Rightarrow T=0|S$
(if i>0)

87

87

# 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

88

88

# 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
```

89

89

# Summary and history

90

## Functional dataflow summary

- We have introduced a simple and expressive paradigm for concurrent programming
  - We can build multi-agent programs using streams (lists with unbound tail) and agents (list functions 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)
  - Functional dataflow is a form of functional programming
  - « Concurrency for Dummies » is the best concurrent paradigm

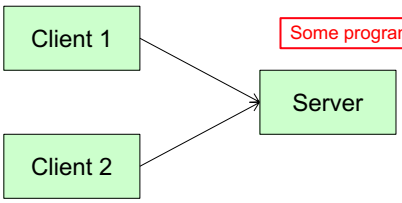
91

## The future: making concurrency simple

- Parallel programming has finally arrived (a surprise to old timers like me!)
  - Multicore processors: quadcore, octocore, 24 cores today, a hundred in a decade, many apps take advantage of it
  - Distributed computing: bigdata analytics with tens of nodes today (NoSQL, MapReduce), hundreds and thousands tomorrow, most apps will do it
- Programs are getting more and more concurrent
  - Sequential programming can't be the default (it's a bottleneck)
  - Libraries cannot hide the issue (interface complexity, distribution structure)
- Concurrency is getting easier
  - "Concurrency for Dummies" is the best paradigm (functional dataflow)!
  - It can be used easily on multicore and distributed systems
    - High performance becomes easy
    - Network transparency (program code is the same for different numbers of cores)
    - Modular fault tolerance is easy

92

# But is determinism the right default? Yes!




Some programs cannot be written in functional dataflow

A client/server can't be written in functional dataflow!

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

- Functional dataflow has strong limitations!
  - A program that needs nondeterminism can't be written
  - Even a simple client/server can't be written

Can be solved! Just add nondeterminism exactly where it is needed

- But determinism has enormous advantages, so it is the correct default
  - Race conditions are impossible
  - With determinism as default, we 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

93

# History of functional 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? Wrong!
- 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).

94

# LINFO1104: Lecture 10
# Limitations of functional dataflow

95

95

## Overview of lecture 10

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

96

96

# Limitations of functional dataflow (part 1)

97

97

## Limitations of functional dataflow

- In lectures 9 we saw functional dataflow, which makes concurrent programming very easy
  - It allows “Concurrency for Dummies”: threads can be added to a 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.

98

98

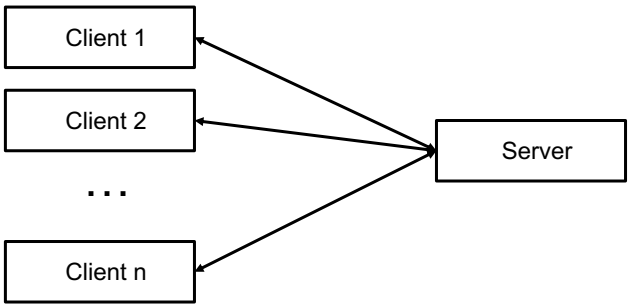
# Client/server application (1)

- 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 functional dataflow!
  - Why not? Let's try and see what goes wrong! Try it yourself!

99

99

# 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

100

100

# Client/server: first attempt

- Let's try to write a client/server in functional 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
```

101

101

# 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 does not!

102

102

## 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?)

103

103

## 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!

104

104

# 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

105

105

# Overcoming the limitations

106

106

## Overcoming the limitations

- Functional 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

107

107

## 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} returns
- 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!)

108

108

# 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?

109

109

# 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 nice!

110

110

# 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 P $M_i$} for each message it sends
  - The server reads the stream S, which contains all messages from all clients in some nondeterministic order (n-to-1 channel)

111



# 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?

112

# 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)

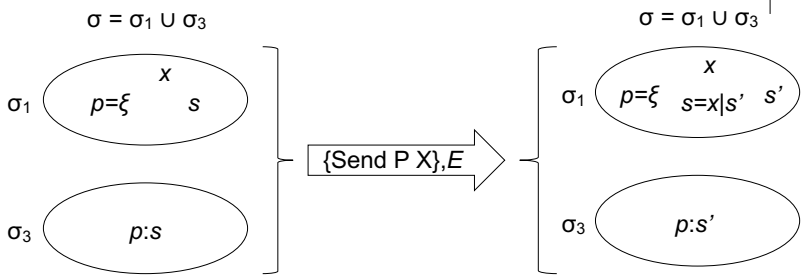
environment

- 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 $\sigma_3$
- {Send P X}, {P→*p*, X→*x*}
  - Assume *p*=ξ, unbound variable *s* ∈ $\sigma_1$ , *p*:*s* ∈ $\sigma_3$
  - Create fresh unbound variable *s'*, bind *s*=*x*|*s'*, update pair to *p*:*s'*

113

113

# Port semantics (2)




- {Send P X} 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*}

114

114

## Client/server with ports

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

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

115

115

## Message-passing concurrency

116

116

# Message-passing concurrency

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

117

117

# Stateless port objects (stateless agents)

118

118

## 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!

119



## A math agent

- Here is a simple procedure to do arithmetic:

```
proc {Math M}
   case M
   of add(N1 N2 A) then A=N1+N2
   [] mul(N1 N2 A) then A=N1*N2
   …
   end
end
```

120

# 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
```

121



# 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
```

122

## 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
```

123

123

## 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

124

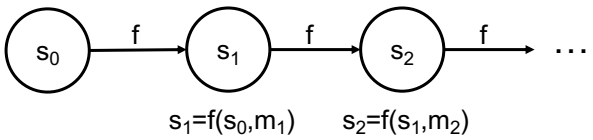
124

# Stateful port objects (stateful agents)





## 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

# 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
```

127



# 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
- …

128

## 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!

129

129

## 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
```

130

130

## 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…

131

131

## 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

132

132

# Example Cell agent

- This transition function behaves like a cell!

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

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

133

133

# 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
```

- Let's say we want the same interface as objects:

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

134

134

# 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 either port objects or standard objects as we saw before

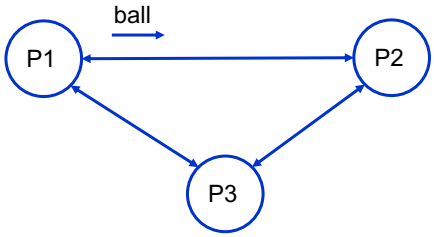
135



# Play ball example

136

# 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!

137

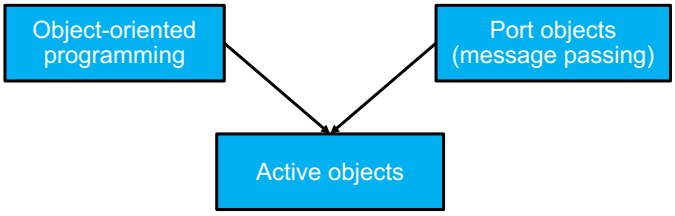
137

# Active objects

138

138

# 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

139

139

# 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
```

140

140

# 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
```

141

141

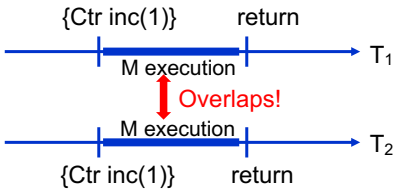
# Passive objects and active objects

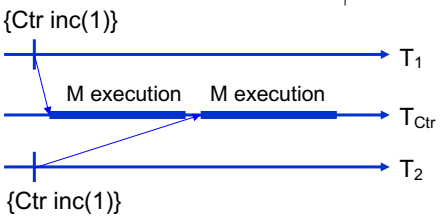
- We make a distinction between passive objects and active objects
- Standard objects in Oz (and many other languages, such as Java and Python) are now called passive objects
  - This is because they execute in the caller's thread; they do not have their own thread
- This is in contrast to active objects,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

142

142

# 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

143

143

# 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

144

144

# Message protocols

145

145

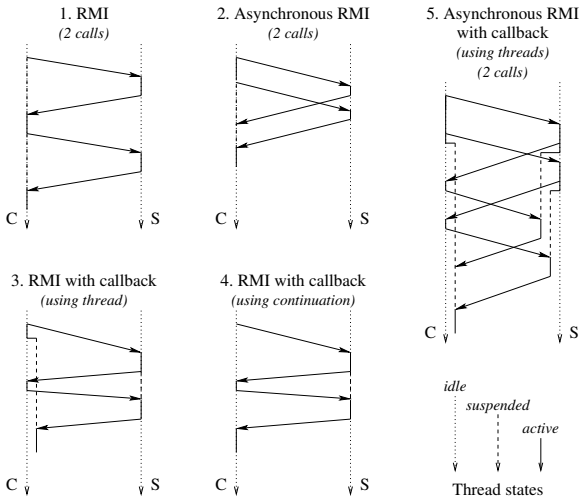
## Message protocols (1)

- 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

146

146

# 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

147

147

# Functional dataflow with ports

148

148

# The best way (as far as I know)

- Writing general concurrent programs is difficult!
  - But functional dataflow is easy (“Concurrency for Dummies”)
  - Can functional dataflow offer help for general programs? Yes!
- This leads to the best way to write concurrent programs
  - Start with functional 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

149

149

# Concurrent composition (fixed number of threads)

150

150

# 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
```

151

151

# 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 what order we wait

152

152

# Higher-order abstraction

- Using higher-order programming, we implement the general form:
  $(\langle s\rangle_1 \,||\, \langle s\rangle_2 \,||\, \ldots \,||\, \langle s\rangle_n)$
- The instruction $\langle s\rangle_1$ is written as **proc** {$} $\langle s\rangle_1$ **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 functional dataflow only
  - No ports needed; we will add one port later when we make it dynamic

153



# 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

154

# 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** ]}

155



# Concurrent composition (variable number of threads)

156

# 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!
    - In the new thread created by **thread** <s> **end**, the <s> can also create threads
- This abstraction cannot be written in functional 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 functional dataflow, with just one added port for doing one specific nondeterministic thing.

157

157

# Specifying the abstraction

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

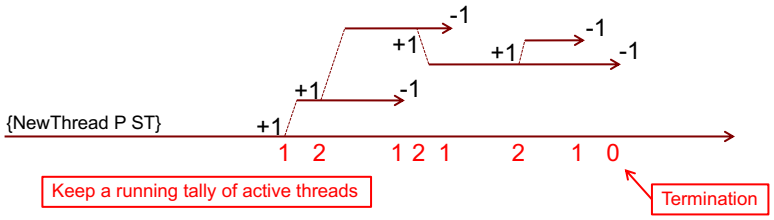
```
{NewThread proc {$} <s> end SubThread}
{SubThread proc {$} <s> end}
```

- 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

158

158

# 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

159

159

# 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
```

160

160

# 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

161

161

# 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!

162

162

# 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





# 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

## 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
```

165

165

## Eliminating sequential dependencies

166

166

# 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 functional 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

167

167

# Example: 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!

168

168

# 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

169

169

# 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!)

170

170

# 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**

171

171

# 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

172

172

# Conclusion





# Conclusion: how to build concurrent programs

- We have seen three good paradigms for concurrent programs
- Functional dataflow (including lazy): best paradigm, with 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): less good, but without limitations
  - Stateful agents that communicate with asynchronous messages
  - **Erlang** is a successful industrial example of this approach
- Functional dataflow with ports: best paradigm without limitations
  - Write most of the program as functional 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!

# LINFO1131: Lectures 2 & 3 Lazy evaluation and declarative programming





## Overview

- Introduction to lazy evaluation
  - Semantics based on dataflow
- Lazy streams
  - Producer-consumer in four paradigms
  - Infinite lists
  - Hamming problem
- Lazy suspensions
  - Graphical representation of lazy evaluation
- Lazy functional dataflow
  - Bounded buffer
- Lazy quicksort
  - Inventing an incremental algorithm
- What is declarative programming?
  - Partial termination
  - Equivalent stores
  - Introduction to first-order logic
  - Definition of declarative programming
  - Failure confinement
- Table of declarative paradigms
- Conclusions

# Introduction to lazy evaluation





## 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}
```

# Semantics of lazy evaluation





# 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…

# 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}

181

181

# 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}

182

182

# 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

183

183

# Producer-consumer in four paradigms

184

184

# Producer-consumer pipeline

- We give the code of a simple producer-consumer pipeline
  - We will run the code in four different functional paradigms
  - All four paradigms are declarative and end up with the same result
  - But the result appears in four different ways
- 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
```

185

185

# Four functional paradigms

As seen earlier

1. Sequential functional programming
   - No single assignment
   - Traditional functional languages (Lisp, Scheme, ML, OCaml)
2. Sequential FP with single assignment
   - Default execution of functions in Oz
3. Eager functional dataflow
   - Adds threads and dataflow synchronization
   - Multi-agent programming with declarative agents

New!

4. Lazy evaluation (introduced in this lecture!)
   - Adds by-need synchronization
   - Lazy functional languages (Haskell, Miranda)

186

186

# 1. Sequential FP (no single assignment)

- We generate 10 elements
  - This is traditional FP with no single assignment
  - Nothing is displayed until after 10 seconds
    - S1 and S2 both displayed at once after 10 seconds
  - Both S1 and S2 are created as a batch execution
    - Why?

```
declare S1 S2 in
{Browse S1}
{Browse S2}
S1={Prod2 1 10}
S2={Cons S1 0}
```

```
% Prod2 uses no unbound variables
fun {Prod2 L H}
   {Delay 1000} % Wait 1000 ms
   if L>H then nil
   else S in
      S={Prod2 L+1 H}
      L|S % Both L and S are bound
   end
end
```

187

187

# 2. Sequential FP with single assignment

- We generate 10 elements
  - S1 is augmented every second
  - S2 is not displayed until after 10 seconds
  - S1 is created incrementally
    - Why?
    - Translate to kernel language!
  - S2 is created as a batch
    - Why?

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

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

188

188

# 3. Eager functional dataflow

- We execute both calls in their own threads
  - This is running functional dataflow (eager)
  - What is the difference with the previous version?
    - Both S1 and S2 are created incrementally
    - Each thread is a declarative agent

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

189



# 4. Lazy evaluation

```
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} needs the second element of S2, which will activate its computation and display it

190

# 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

191

191

# Infinite lists

192

192

# Infinite lists

- With lazy evaluation we can compute with infinite loops
  - We can write programs with infinite lists
  - It works because the execution only computes needed elements
  - This is not possible in eager functional dataflow!
- 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}

193

193

# 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

194

194

# 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}
```

195

195

# Hamming problem

196

196

# 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.
  - Richard Hamming. *The Art of Doing Science and Engineering: Learning to Learn.* 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

197

197

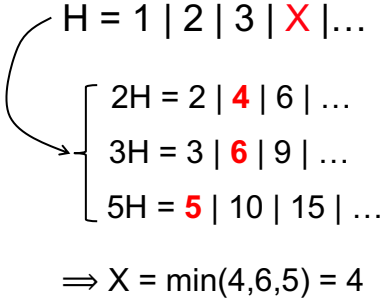
# Hamming problem

- 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 compute them incrementally until we are satisfied
  - The program should be efficient in time and memory!

198

198

# Algorithm idea




- 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

199

199

# 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

200

200

# Hamming program

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

- Main expression:

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

```
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
```

201

201

# Lazy suspensions

202

202

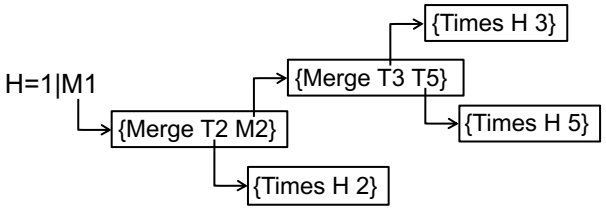
# Lazy suspensions

- 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 → {Times L1 3}
  "A thread is suspended on L2 that contains the body of {Times L1 3}"

203

203

# 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'

204

204

# 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**
    Activate {Times H 2}
    Activate {Merge T3 T5}
  - 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

205

205

# Next activations

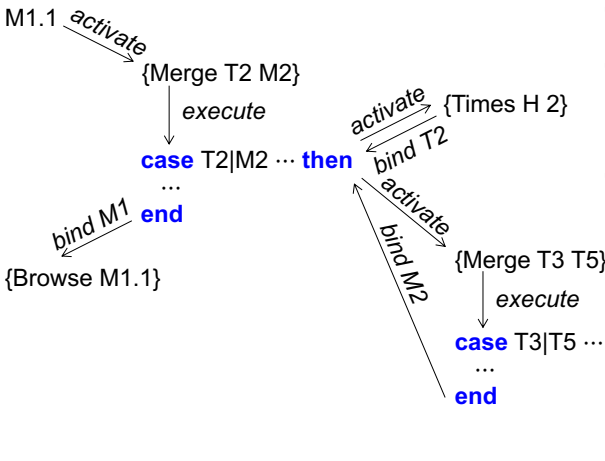
- {Times H 2} and {Merge T3 T5} are activated
  - The body of {Times H 2} is executed
    - This binds T2=2|T2' and creates a new lazy suspension on T2': {Times M1 2}
  - The body of {Merge T3 T5} is executed
    - This activates {Times H 3} and {Times H 5}
    - After executing these two functions, 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'

206

206

# Overall execution flow

The execution is sequential (follow the arrows!)




- 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

207

207

# Lazy functional dataflow

208

208

# Five functional paradigms

- So far we have seen four paradigms of functional programming:
  - Sequential functional programming (no single assignment)
    - Traditional functional languages do this (Lisp, Scheme, ML, OCaml)
  - Sequential functional programming with single assignment
    - Allows data structures with "holes", e.g., list functions are tail-recursive
    - This is the default way that Oz executes functions
  - Eager functional dataflow
    - Adds threads and dataflow synchronization
    - Allows concurrent programming with streams (multi-agent programming)
  - Lazy evaluation
    - Adds by-need synchronization (with WaitNeeded), where functions are executed only when their results are needed
    - Allows programming with infinite data structures
    - Lazy functional languages do this (Haskell, Miranda)
- There is a fifth paradigm:
  - Lazy functional dataflow
    - Adds both threads and lazy functions

209

209

# Lazy functional dataflow

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

210

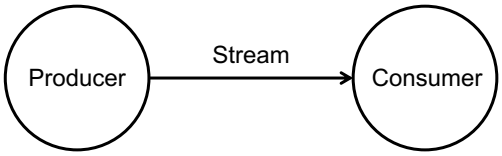
210

# Bounded buffer

211

211

# Bounded buffer (1)

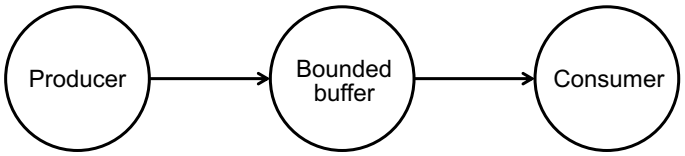



- A producer-consumer pipeline has performance problems
  - Variations in producer and consumer speeds can cause the system to perform poorly
  - When a producer creates elements too quickly, the consumer cannot use the elements so the producer idles
  - When a consumer needs more elements, the producer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

212

212

# Bounded buffer (2)

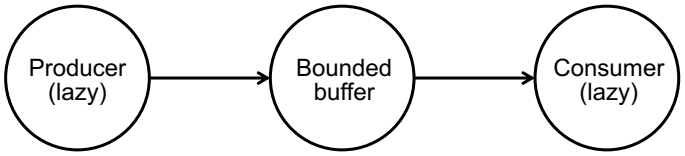



- Inserting a bounded buffer can solve these problems
  - When the producer creates elements too quickly to be consumed, they are stored in the bounded buffer
  - When the consumer needs more elements than can be produced, they are taken from the bounded buffer
  - This improves performance by smoothing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

213

213

# 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

214

214

# 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

215

215

# 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
```

216

216

# 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

217



# 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
```

218

# 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 functional dataflow, 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

219

219

# 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

220

220

# Lazy quicksort

221

221

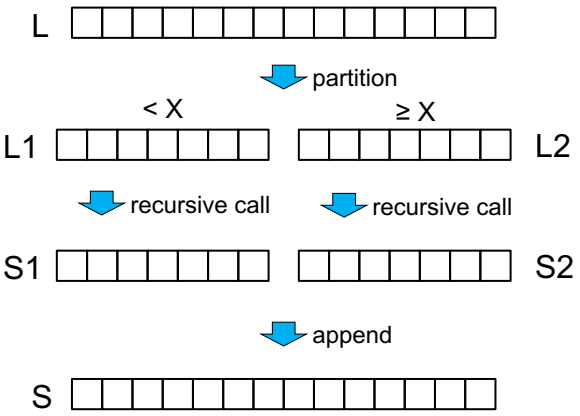
## Lazy quicksort

- 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!

222

222

# Quicksort algorithm




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

223

223

# 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]

224

224

# 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
```

225

225

# 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
```

226

226

# 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!

227



# 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

228

# 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

229

229

# 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
```

230

230

# 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?

231



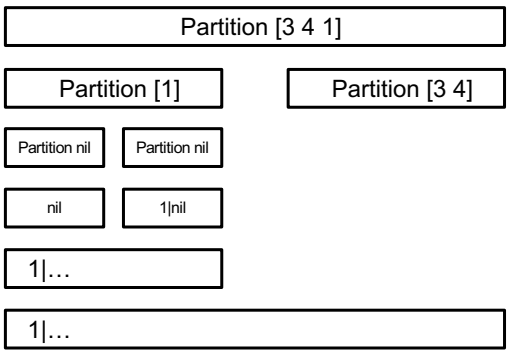
# Execution steps…

- 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}

232

# Complexity of lazy quicksort




n traversal

n/2 traversal
(n/4, n/8, continues to 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)

233

233

# What is declarative programming?

234

234

# Declarative programming

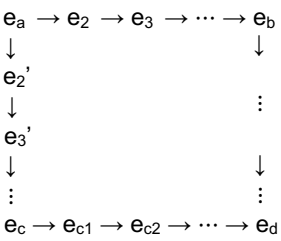
- We have seen five functional paradigms
  - Sequential functional programming
  - Sequential functional programming with single assignment
  - Functional dataflow (concurrent)
  - Lazy evaluation
  - Lazy functional dataflow (concurrent)
- We claim that they are all declarative
  - What does this mean, 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)

235

235

# Functional programming

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




- 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

236

236

# 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? (threads and their scheduler)
  - Streams? (programs that 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

237

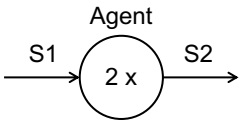
237

# I: Partial termination

238

238

# 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?
  - Problems: (1) the program never terminates and (2) 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?
  - See next section!

239

239

# II: Equivalent stores

240

240

# 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?
  - We first need to introduce some concepts from formal logic → Intro slides

241

241

# 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 all values of *x*, *y*, *z*, and *w*, there are two possibilities: either they can be part of a store σ or they cannot be part of a store σ
  - So the store σ is a logical formula, which can be true or false
  - We write σ as a logical formula: σ ≡ *x*=foo(*z w*) ∧ *y*=foo(*z w*)

242

242

# 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 $\alpha \models \beta$ which means "β is true in all models of α"
- Definition: Two stores σ and σ' are logically equivalent if
  - $\sigma \models \sigma'$ and $\sigma' \models \sigma$ σ' is **true** in all models of σ and σ is **true** in all models of σ'
- Another way to write this is:
  - $\models (\sigma \Leftrightarrow \sigma')$ $(\sigma \Leftrightarrow \sigma')$ is a tautology, i.e., it is **true** in all models

243

243

# Introduction to first-order logic

244

244

# First-order logic

- To define store equivalence, we introduce first-order logic
- Formal logic is:
  - A formal language: a syntactic definition of a set of formulas
  - A proof theory: a set of rules to deduce whether a formula is true or false, given a set of primitive formulas (axioms)
  - A model theory: mathematical objects in which the axioms are true
- First-order logic is:
  - A formal logic with variables, quantifiers, predicates, and connectors
    - Axiom: $\forall x.\forall z.(\text{grandparent}(x,z) \Leftrightarrow \exists y.\text{parent}(x,y) \wedge \text{parent}(y,z))$
    - Model: any set of human beings with the parent relation
  - Some popular programming languages based on first-order logic are Prolog, constraint programming, and SQL

245

245

# Example of first-order logic

- Formulas:
  syntactic expressions
  (with variables and quantifiers)

$\forall x.\ x<x+2$

$\exists x.\ (x^2-3x+2=0)$

$\forall x.\ (x^2-3x+2=0 \Rightarrow x=1 \vee x=2)$

$\forall x.\ \exists y.\ y=x^2$

- Models:
  integers $\mathbb{Z}$
  $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$

  reals $\mathbb{R}$
  $\mathbb{R} = \{x \mid x \text{ has infinite decimal expansion}\}$

- All formulas on the left are true in $\mathbb{Z}$ and $\mathbb{R}$

246

246

# Example of proof and model

- Given a set of formulas F
  - Given any formulas α, β ∈ F
- Proof theory
  - Given a set of proof rules
  - **α ⊢ β** : β can be deduced from α using the rules
  - $(\exists x.x^2=1) \vdash \exists x.(x=1 \vee x=-1)$
- Model theory
  - **α ⊨ β** : β is true in all models in which α is true
  - If $(\exists x.x^2=1)$ true in $\mathbb{R}$ then $\exists x.(x=1 \vee x=-1)$ true in $\mathbb{R}$

247

247

# Model of a store σ

- We give a formal definition of a model of a store σ
- First step: An interpretation of a store σ (is true or false)
  - 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**
- Second step: A model of a store σ (is always true)
  - A model of σ is an interpretation in which σ is **true**

248

248

# III: Definition of declarative programming

249

249

## 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

250

250

# IV: Failure confinement

251

251

## 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

252

252

# 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

253

253

# 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
```

254

254

# Table of declarative paradigms





# Declarative paradigms according to the textbook

| | *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

# Conclusions

257

257

# Conclusions

- Lazy evaluation
  - Functions are evaluated only if their results are needed
  - This extends dataflow (Wait & Bind) 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 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

258

258

# LINFO1131: Lecture 4 Advanced declarative algorithm design

259

259

## Overview

- Motivation
  - Writing efficient algorithms in declarative paradigms
- Concepts used
  - Single assignment and lazy evaluation
  - Amortized and worst-case
  - Ephemeral and persistent
- Summary of the algorithms
- Difference list
  - A list representation with efficient ephemeral operations
- Naïve queue
- Amortized constant-time ephemeral queue
- Worst-case constant-time ephemeral queue
  - A short step to logic programming
- Amortized constant-time persistent queue
- Worst-case constant-time persistent queue
- Conclusions

260

260

# Motivation

- Writing declarative applications
  - We would like to write as much of the program as possible in a declarative paradigm
- Writing algorithms in a declarative paradigm
  - An important question: is it possible to make efficient algorithms in declarative paradigms?
    - Is mutable state really needed for efficiency? Not always!
  - Declarative algorithms can often be efficient
    - We give many techniques for doing this
    - We use the declarative paradigms we saw before

261



# Recommended book

- Many of the examples in this lecture are taken from the book
  - Chris Okasaki. *Purely Functional Data Structures*.
- This book shows how to use lazy evaluation to define many efficient declarative algorithms
  - The book uses the Standard ML language with an explicit lazy evaluation operation

262

# Concepts used

263

263

## Concepts used

- Today's lecture is based on the following concepts

- Language concepts
  - Single-assignment variables
  - Lazy evaluation

  } We saw these in the previous lecture
- Complexity concepts
  - Amortized upper bound
  - Worst-case upper bound
- Algorithm concepts
  - Ephemeral data structure
  - Persistent data structure

264

264

# Amortized and worst-case

- Amortized complexity
  - If n operations have a combined complexity of O(f(n)) then we say each operation has an amortized complexity of O(f(n)/n)
  - This is important when individual operations are sometimes expensive but operations are cheap on average
  - For example, individual operations on a queue can have complexity O(n), but with n operations it is possible for individual operations to have amortized complexity of O(1)
- Worst-case complexity
  - This is the big-O notation that you have seen before
- A good worst-case upper bound is best, but if this is not possible, an amortized bound may be good enough

265

265

# Ephemeral and persistent

- An ephemeral data structure can have only one version exist at the same time
  - Given queue Q1, then doing Q2={Insert Q1 a} creates a new queue Q2 and Q1 can no longer be used
  - Stateful data structures (like in Java) are always ephemeral: when you change the attributes of an object, the old values are forgotten
- A persistent data structure can have many versions exist at the same time
  - Given queue Q1, then doing Q2={Insert Q1 a} creates a new queue Q2 and Q1 can still be used. Doing Q3={Insert Q1 b} will create another version, and all versions Q1, Q2, Q3 are still usable
  - What are persistent data structures good for?

266

266

# The use of persistence

- Persistent data structures are used for collaborative work
  - Two people edit the same text and create their own versions
  - Software repositories like github support multiple versions
  - Some databases support multiple versions
- In systems with mutable state (like Java), multiple versions can be simulated by doing explicit “copy” operations, but this is clumsy
  - In our declarative code, the versions are handled automatically
- “Merge” operation
  - Persistent data structures often have a “merge” operation that allows to combine two versions
    - The merge takes care of possible conflicts between versions

267

267

# Summary of the queues

268

268

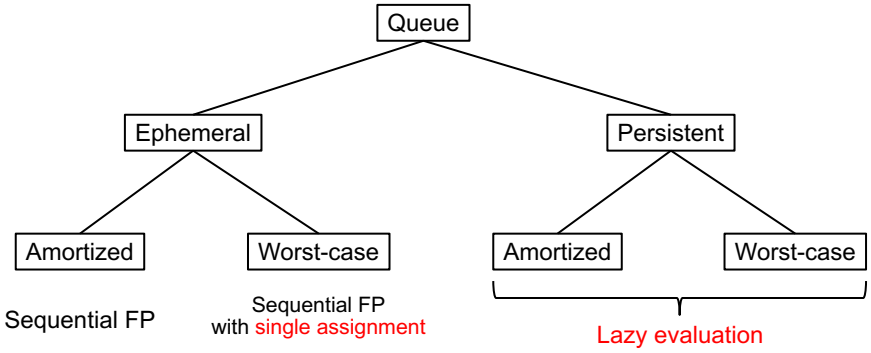
# Summary of the queues

- We show five ways to implement a queue abstraction
  - We first give a naïve algorithm; the others have better properties!
  - We use three declarative paradigms for these implementations
- Sequential functional programming
  - Naïve queue, insert is O(1) and delete is O(n)
  - Amortized ephemeral queue, insert and delete O(1)
- Sequential functional programming with single assignment
  - Worst-case ephemeral queue, insert and delete O(1)
- Lazy evaluation
  - Amortized persistent queue, insert and delete O(1)
  - Worst-case persistent queue, insert and delete O(1)

269

269

# Summary of the queues




270

270

# Difference list





# Difference list

- A difference list is a representation of a list as a pair of two lists (S,E) such that E is a suffix of S
- The difference list (S,E) represents the list S–E where we take S and remove the suffix E
  - The list 1|2|3|nil can be represented as a difference list:
    - As the pair (1|2|3|X,X) where X is an unbound variable
    - As the pair (1|2|3|4|Y,4|Y) where Y is an unbound variable
    - As the pair (1|2|3|4|5|nil,4|5|nil)
    - And so forth, there are an infinite number of possibilities
- What is the advantage of this representation?
  - A difference list has efficient ephemeral operations!

# Adding an element

- Add an element to the start or the end in constant time
  - Given the difference list (S,E) where E is an unbound variable
  - Adding X to the start gives (S1,E) with the binding S1=X|S
  - Adding X to the end gives (S,E1) with the binding E=X|E1
- For example, take the difference list (1|2|3|X,X)
  - Adding 4 to the start gives (4|1|2|3|X,X)
  - Adding 4 to the end gives (1|2|3|4|Y,Y)
    - Bind X=4|Y, then take the first element of the original difference list together with Y, which gives (1|2|3|4|Y,Y)
- For a standard list, like 1|2|3|nil, it is not possible to add an element at the end in constant time!





# Constant-time append

- We append two difference lists in constant time
  - Given (S1,E1) and (S2,E2), then the append is given by (S1,E2) with the binding E1=S2
  - For example, we append (a|b|X,X) and (1|2|Y,Y) by binding X=1|2|Y and taking the start of the first with the end of the second (a|b|1|2|Y,Y)
- This can only be done once (it is ephemeral)!
  - In many cases, that is all we need
  - Let us show an example how it can be used

# Naïve flatten

- Append function

```
fun {Append L1 L2}
   case L1 of nil then L2
   [] H|T then
      H|{Append T L2}
   end
end
```

- Append is a tail-recursive function whose execution time is proportional to |L1|, i.e., the length of L1

- Naïve flatten function

```
fun {Flatten Xs}
   case Xs of nil then nil
   [] X|Xr andthen
     {IsList X} then
      {Append {Flatten X}
         {Flatten Xr}}
   [] X|Xr then
      X|{Flatten Xr}
   end
end
```

- {Flatten [[1 2] [3 4] 5]} gives [1 2 3 4 5]

275



# Flatten with a difference list

- We replace the list result by a difference list
  - We add two arguments S and E
  - We define it as a procedure to show the arguments

```
proc {DFlatten Xs S E}
   case Xs of nil then
      S=E
   [] X|Xr andthen {IsList X} then M in
      {DFlatten X S M}
      {DFlatten Xr M E}
   [] X|Xr then M in
      S=X|M
      {DFlatten Xr M E}
   end
end
```

This is much more efficient than the naïve Flatten!

(S,E) is the append of (S,M) and (M,E)

276

# Difference lists versus linked lists

- These two data structures seem quite similar
  - Difference list (functional)
  - Linked list (stateful, for example in Java)
- What's the difference?
  - Both allow efficiently building chains of elements
  - The difference is that difference lists cannot be broken: they have functional semantics
  - Linked lists can always be broken: the chains can always be modified by assignment

277

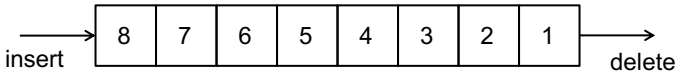
277

# Naïve queue

278

278

# Queue abstraction




- A queue is a sequence that allows insert on one end and delete on the other end
  - Insert 1, 2, …, 8
  - First-in first-out (FIFO): delete 1, 2, …, 8
- We first give a naïve queue implementation in sequential functional programming

279

279

# Naïve queue

- We define a simple queue abstraction
  - Not a true abstraction since the representation is not protected
  - What is its complexity?
- Helper function:

```
fun {ButLast L X}
   case L
   of [Y] then X=Y nil
   [] Y|L1 then
      Y|{ButLast L1 X}
   end
end
```

- Queue operations:

```
fun {NewQueue} nil end
fun {Insert Q X} X|Q end
fun {Delete Q X}
   {ButLast Q X}
end
```

- Example execution:

```
declare Q Q1 1 X1 in
Q={Insert {Insert {Insert
      {NewQueue} 1} 2} 3}
{Browse Q}
Q1={Delete Q X1}
{Browse X1}
```

280

280

# Amortized constant-time ephemeral queue

281

281

## Amortized ephemeral queue

- We define an amortized constant-time ephemeral queue in functional programming
- The queue is represented as a tuple q(F R)
  - Content is {Append F {Reverse R}}
  - Insert is done by updating R
  - Delete is done by updating F
  - If F is empty, then {Reverse R} is done to move elements from R to F

```
fun {NewQueue} q(nil nil) end
fun {Check Q}
   case Q of q(nil R) then
      q({Reverse R} nil) else Q end
end
fun {Insert Q X}
   case Q of q(F R) then
      {Check q(F X|R)} end
end
fun {Delete Q X}
   case Q of q(F R) then F1 in
      F=X|F1 {Check q(F1 R)} end
end
```

282

282

# Discussion

- Example execution: (exactly as before!)
  Q={Insert {Insert {Insert {NewQueue} 1} 2} 3}
  {Browse Q}
  Q1={Delete Q X1}
  {Browse X1}

- Questions
  - If this queue is used in a persistent manner (with multiple versions) then the results will be correct. Why?
  - However, when used in persistent manner, this queue is no longer amortized constant-time. Why?
    - Hint: find a sequence of n operations that has worse complexity than O(n)

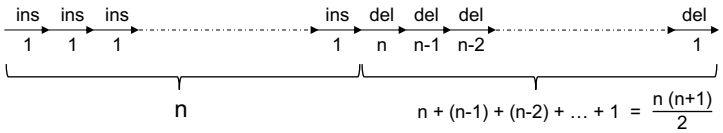
283

283

# Comparing naïve queue with amortized queue

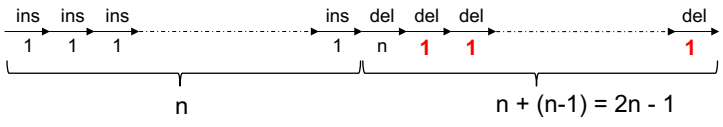
Naïve queue (single list L)

Worst-case O(n)
Amortized O(n)




Amortized constant-time ephemeral queue (two lists F and R)

Worst-case O(n)
**Amortized O(1)**




284

284

# Worst-case constant-time ephemeral queue





# Worst-case ephemeral queue

- It is not possible to write a queue with constant-time insert and delete in standard functional programming
  - Amortized constant-time is the best we can do
  - But adding single assignment makes it possible!
- The queue is represented as the tuple q(S E) where (S,E) is a difference list with the content
- Both insert and delete are always constant-time!

```
fun {NewQueue} X in q(X X) end
fun {Insert Q X}
   case Q of q(S E) then E1 in
      E=X|E1 q(S  E1)
   end
end
fun {Delete Q X}
   case Q of q(S E) then S1 in
      S=X|S1 q(S1 E)
   end
end
```

# Knowing how many elements

- The previous definition does not let us know when the queue is empty
- To know the number of elements, we add the queue size to the representation
  - The queue is represented as q(N S E) where N is the number of elements and (S,E) is the same as before
  - Test if empty:
    **fun** {IsEmpty Q) Q.1==0 **end**
- Both insert and delete are still constant-time!

```
fun {NewQueue} X in q(0 X X) end
fun {Insert Q X}
   case Q of q(N S E) then E1 in
      E=X|E1 q(N+1 S  E1)
   end
end
fun {Delete Q X}
   case Q of q(N S E) then S1 in
      S=X|S1 q(N-1 S1 E)
   end
end
```

287

287

# A short step to logic programming

288

288

# Doing delete before insert

- Try the following execution:
  **declare** Q1 Q2 Q3 X **in**
  Q1={NewQueue}
  Q2={Delete Q1 X} % Delete from an empty queue
  {Browse X}
  Q3={Insert Q2 foo} % Insert an element

- This first displays an unbound variable X
  - When foo is inserted, the display is updated to foo
  - The delete creates an empty slot that is filled later by insert
  - How can this work?

289

289

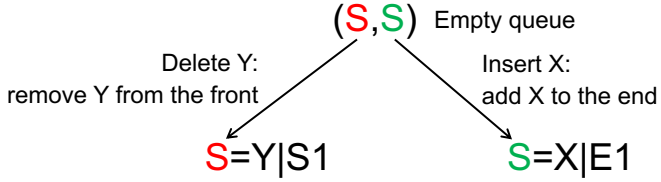
# Special power of this queue

- This queue definition has a special power that follows from the logical equivalence property of stores
  - The queue can have a negative number of elements!
  - An element can be deleted before it is inserted
- The queue definition guarantees that deleted elements are equal to inserted elements
  - It is because binding done in any order gives the same results
    - Binding is a symmetric operation; the general binding operation is called unification and it follows from the logical equivalence of stores
  - We can delete an unbound variable first and insert a value later
- We are doing more than just functional programming
  - We are doing logic programming, similar to what Prolog does

290

290

# Delete before insert explained




- We start with an empty queue (S,S)
- Let's do delete Y and insert X on the empty queue
  - This gives two bindings, S=Y|S1 and S=X|E1
- What happens in the store?
  - We have the logical formula ($s=y|s_1 \wedge s=x|e_1$)
  - Simplifying shows us that $y=x$ and $s_1=e_1$

291

291

# Shoutout to Prolog

- The queue is doing logic programming
  - Both insert and delete do unification with the same variable
  - Because of logical equivalence, this imposes logical equality
  - Remember the last lecture's introduction to first-order logic!
- Logic programming is another declarative paradigm
  - Logic programming is more general than functional programming
  - Data structures are truths: they can have unbound variables and binding is bidirectional (both inputs and outputs can be bound)
  - Computation is deduction: a running program deduces new truths
    - A Prolog program is actually a theorem prover
- If you are curious, check out Prolog and constraint programming
  - See programming paradigms and constraint programming courses

292

292

# Amortized constant-time persistent queue

293

293

## Making it persistent

- Persistence is a strong property that is hard to get
  - As your program updates its data structures, many versions are created that exist simultaneously
  - Stateful programming, like in Java and Python, is ephemeral by default. All algorithms using mutable state are ephemeral by default!
    - Stateful algorithms can be made persistent by making explicit copies, but this is hard because it is managed by the programmer
- Declarative algorithms can be made persistent by using lazy evaluation
  - This is another amazing property of lazy evaluation

294

294

# Helper function: lazy append

- We define a lazy append like we did before with quicksort

```
fun lazy {LAppend Xs Ys}
   case Xs of X|Xr then X|{LAppend Xr Ys}
   [] nil then Ys end
end
```

- Example execution:

```
declare L in
L={LAppend [1 2 3] [4 5 6]}
{Browse L}
```

- Run this and ask for elements of L, to understand it!
  - It gives 1, 2, 3, and then [4 5 6] all at once

295



# Persistent algorithm idea

- We define the queue again, but with yet another representation
- We use a tuple q(LenF F LenR R) where LenF and LenR are integers giving the length of F and R
  - As before, we move elements from R to F when F becomes empty
  - But now we do the move with a lazy suspension
- How we get amortized constant-time
  - The move does a {Reverse R} which cannot be made incremental
  - To make it amortized, we pay for the lazy suspension in advance
  - We use the "banker's method": we do n operations in advance before creating the lazy suspension
  - It is like saving money: save bit by bit and buy when you have enough

296

# Persistent algorithm code

```
fun {NewQueue} q(0 nil 0 nil) end
fun {Check Q}
   case Q of q(LenF F LenR R) then
      if LenF<LenR then
         q(LenF+LenR {LAppend F {fun lazy {$} {Reverse R} end}} 0 nil)
      else Q end
   end
end
fun {Insert Q X}
   case Q of q(LenF F LenR R) then {Check q(LenF F LenR+1 X|R} end
end
fun {Delete Q X}
   case Q of q(LenF F LenR R) then F1 in F=X|F1 {Check q(LenF-1 F1 LenR R)} end
end
```

*Move R to F (lazily)*

*Increase R*

*Decrease F*

297

297

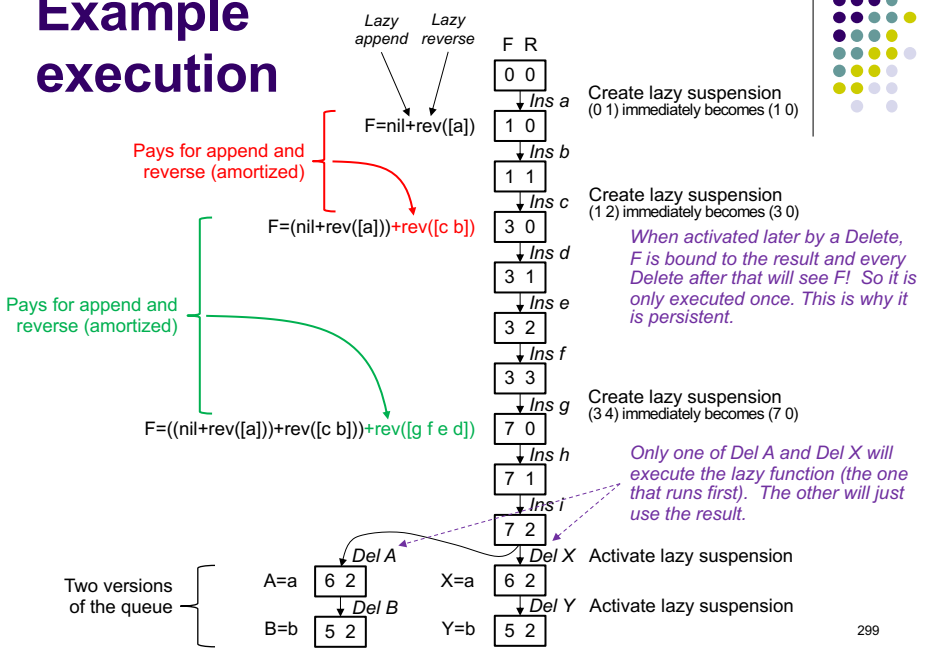
# How it works (example on next slide)

- The trick is to make sure the algorithm creates a lazy suspension at the right time
  - It has to be done when the {Reverse R} is paid for
    - Banker's method: save operations in the bank!
  - Assume we are inserting elements
    - This causes R to increase
    - When R is larger than F, create the lazy suspension
    - Activating the lazy suspension makes F bigger and R empty
  - Assume we are deleting elements
    - We activate a lazy suspension to get an element, and this triggers the {Reverse R}, but it's ok since it's paid for

298

298

# Example execution




299

299

# Why {Reverse R} is monolithic

- The {Reverse R} function cannot be made incremental by lazy evaluation
  - We say that it is monolithic
- It is because we cannot know the first element of the reversed list without traversing the whole list
  - Any function where we need to see the whole data structure in order to create a single output cannot be made incremental by lazy evaluation (another example: Partition in lazy quicksort)
- We show the code…
  - If you try to execute Reverse lazily you will see why this happens: the recursive calls of Reverse don't create any results until the recursion stops at the end

300

300

# Execution of lazy reverse

- Reverse function:

```
fun lazy {Reverse L A}
   case L of X|L2 then
      {Reverse L2 X|A}
   [] nil then A
   end
end
```

- Traverse list L and build reverse in accumulator A
- Sample call: R={Reverse [1 2 3] nil}
- What happens when we ask for the first element? {Browse R.1}
- R is needed so the lazy suspension is activated and executes the body. This calls: {Reverse [2 3] 1|A}
- This creates another lazy suspension that is immediately activated because it is needed!
  - To get the first element, we keep traversing L to the end

301

301

# Worst-case constant-time persistent queue

302

302

# Achieving worst-case constant-time

- The reason why the previous example was amortized constant-time was because of the {Reverse R} call
  - Reverse is monolithic: it is executed all at once
- To fix this, we need to execute the reverse step by step
  - The old code is {LAppend F {**fun** lazy {$} {Reverse R} **end**}}
    - This code does both LAppend and Reverse
  - The trick is to merge them into a new function AppRev
    - Each time LAppend does one iteration, we do one step of Reverse
    - The execution of Reverse is "spread out" over n operations
- We show how to merge Append and Reverse

303

303

# "Spreading out" the Reverse

- Old code: {LAppend F {**fun** lazy {$} {Reverse R} **end**}}
  - This code will first get elements lazily from F
  - When F is completely used up, then it executes {Reverse R}
  - It calculates all elements of {Reverse R} in one operation
- New code: {LAppRev F R nil}
  - The function LAppRev is like LAppend, but whenever it does one iteration of Append, it also does one iteration of Reverse
  - When the LAppRev is done (because F is completely used up), then the Reverse is completely executed!

304

304

# Defining LAppRev

- We explain how we combine Append and Reverse
- Here is the code for Append and Reverse:

```
fun lazy {LAppend F B}
   case F of X|F2 then
      X|{LAppend F2 B}
   [] nil then B end
end
fun {Reverse R B}
   case R of Y|R2 then
      {Reverse R2 Y|B}
   [] nil then B end
end
```

- Here is the code for LAppRev that does both Append and Reverse:

```
fun lazy {LAppRev F R B}
   case pair(F R)
   of pair(X|F2 Y|R2) then
      X|{LAppRev F2 R2 Y|B}
   [] pair(nil [Y]) then Y|B
   end
end
```

- Notes:
  - **Green** arguments come from LAppend, **red** ones from Reverse
  - When F is empty, then R has one element left (due to F<R condition: R has grown bigger than F)

305

305

# Persistent algorithm code (new version)

```
fun {NewQueue} q(0 nil 0 nil) end
fun {Check Q}
   case Q of q(LenF F LenR R) then
      if LenF<LenR then
         q(LenF+LenR {LAppRev F R nil} 0 nil)
      else Q end
   end
end
fun {Insert Q X}
   case Q of q(LenF F LenR R) then {Check q(LenF F LenR+1 X|R} end
end
fun {Delete Q X}
   case Q of q(LenF F LenR R) then F1 in F=X|F1 {Check q(LenF-1 F1 LenR R)} end
end
```

New code replaces old code

*Move R to F (lazily)*

*Increase R*

*Decrease F*

306

306

# Conclusions





# Conclusions

- We define important algorithm concepts
  - Amortized complexity: single operations may be expensive but on average they are efficient
  - Persistence: multiple versions of data structures can be used
- We write efficient algorithms in declarative paradigms
  - We take a simple algorithm, a queue, and show four ways how it can be implemented efficiently
  - We use both lazy evaluation and single assignment
- As a bonus, we make a step toward logic programming
  - Because of logical equivalence of stores, variable binding is actually a symmetric operation called unification
  - Logic programming is the most powerful form of declarative programming – check out Prolog and constraint programming

# Take-away intuitions

- Concepts for efficient declarative algorithms
  - Single assignment for fast ephemeral algorithms
  - Lazy evaluation for fast persistent algorithms
- Why it works
  - Single assignment is a weak form of mutable state that is still declarative but is strong enough for ephemeral algorithms because they only do assignment once
  - Lazy evaluation lets expensive operations be done in advance (which improves behavior for multiple versions) and be decomposed into small steps (which improves behavior for worst-case)





# LINFO1131: Lecture 5 Limitations of declarative programming





## Overview

- Limitations of declarative programming
  - Declarative paradigms are based on lambda calculus: they are confluent but they do not interact with the real world during their execution
  - We explain how to extend declarative paradigms to interact with the real world, by adding imperative concepts such as mutable state or communication channels
- Cells
  - A form of mutable state that allows to overcome the limitations of declarative programming
  - This leads to shared-state concurrency
- Ports
  - A communication channel that allows to overcome the limitations of declarative programming
  - This leads to message-passing concurrency (multi-agent programming)

# Limitations of declarative programming (part 2)

313

313

## Beyond declarative programming?

- Up to now we have seen only declarative paradigms
  - Sequential functional programming
  - Functional dataflow and lazy functional dataflow
  - Efficient declarative algorithms
  - These are powerful and useful paradigms!
- Ideally, your program should be completely declarative!
  - Correctness, testing, and maintenance are much simplified!
  - But unfortunately this is impossible
  - Why is it impossible? Let us see by looking at lambda calculus!
  - Luckily most programs only need a few nondeclarative bits, so most of the program can still be declarative

314

314

# Declarative execution = lambda execution

- Declarative execution is equivalent to lambda execution:

$$e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \ \ldots \rightarrow e_{n-1} \rightarrow e_n$$

- Execution starts with initial expression $e_0$ and reduces it in steps, ending with final expression $e_n$
  - Lambda calculus is Turing complete, it can do all computations
  - The power is a consequence of the Church-Rosser theorem (confluence): final result $e_n$ is independent of the reduction order
- How does this execution interact with the real world?
  - In declarative programming, it does not! All information is already in the initial expression $e_0$, nothing is added later.

315

315

# Interacting with the real world is not declarative

- Practical programs take time to execute
  - Each reduction step $e_{i-1} \rightarrow e_i$ takes time because execution happens on a computer, a physical artifact in the real world
- Some reduction steps interact with the real world
  - They accept inputs (like $s_0$) or they generate outputs (like $s_1$)

$$\begin{array}{ccccccccccc} & & s_0 & & & & s_1 & & & & \\ & & \downarrow & & & & \uparrow & & & & \\ e_0 & \rightarrow & e_1 & \rightarrow & e_2 & \rightarrow & e_3 & \rightarrow & e_4 & \rightarrow \ \ldots \rightarrow & e_{n-1} \rightarrow e_n \end{array}$$

- This is not a lambda execution any more!
  - Because input $s_0$ affects the value of $e_2$, and because output $s_1$ comes from $e_4$ so it is not a lambda final expression

316

316

# Lambda calculus: confluent but no real-world interaction

✓ **Confluent reduction of an initial expression to a final result**
This has very strong mathematical properties that we can use

- For reasoning, debugging, testing, optimization, and maintenance
- For concurrency, parallelism, and distribution
- There is no efficiency penalty compared to other paradigms!

✗ But it can't interact with the real world! Let's see why:

- During the execution, we would like to accept inputs coming from the real world and outputs going back to it
- Declarative programming can't interact with the real world because its execution is a step-by-step reduction of an initial expression to a final result. Reduction steps take time, and the inputs will arrive during this time. The reduction can't use them unless we could put them in the initial expression. But we can't do this, because the inputs are not known in advance.

317

317

# Imperative programming

- To interact with the real world, we need to add something to the declarative paradigms
  - A way to receive inputs and send outputs during execution
  - This is usually called imperative programming
- This lets us interact with the real world, but we also have to give up the goodness of declarative programming
- Can we have our cake and eat it too? Both the good properties of declarative programming and interaction with the real world?
  - No we can't! So what can we do…?

318

318

# The right way to design programs

- Write most of the program in a declarative paradigm
  - And add small pieces of imperative programming only in those places that interact with the real world
  - Usually there are only a very few such places, so we keep most of the advantages of declarative programming
- We can use this to improve existing systems too…
  - Existing systems are often not designed like this! They do too much imperative programming. Older systems like Java are especially bad.
  - This gives us a measure to judge how well existing systems are designed (and a way to improve them: make them more declarative)

319



# Kinds of interactions

- There are many ways that a program can interact with the real world
- Here are three typical possibilities:
  - Hardware clock: Input $s_k$ gives the clock time
  - Mutable state: output $s_a$ writes to a register, later input $s_b$ (with b>a) reads from the register
  - Communication channel: output $s_b$ sends to a channel, later input $s_c$ receives from the channel
- Executions give different results depending on the exact timing and order of the reductions

320

# Hardware clock

- Here is an example of a hardware clock:

$$\begin{array}{ccccccccccc} & & t_0 & & t_1 & & t_2 & & & & \\ & & \downarrow\text{read} & & \downarrow\text{read} & & \downarrow\text{read} & & & & \\ e_0 & \rightarrow & e_1 & \rightarrow & e_2 & \rightarrow & e_3 & \rightarrow & e_4 & \rightarrow & e_5 \rightarrow \ldots \rightarrow e_{n-1} \rightarrow e_n \end{array}$$

- A read returns the current time from the clock
  - Reduction $e_0 \rightarrow e_1$ reads time $t_0$
  - Reduction $e_2 \rightarrow e_3$ reads time $t_1$
  - Reduction $e_4 \rightarrow e_5$ reads time $t_2$
- Exact time values depend on reduction timing and order
  - This is not lambda reduction, since for a lambda reduction the result is independent of reduction timing and order (confluence)

321

321

# Mutable state

- Here is an example of a mutable variable:

$$\begin{array}{ccccccccccc} & & x & & y & & y & & & & \\ & & \uparrow\text{write} & & \uparrow\text{write} & & \downarrow\text{read} & & & & \\ e_0 & \rightarrow & e_1 & \rightarrow & e_2 & \rightarrow & e_3 & \rightarrow & e_4 & \rightarrow & e_5 \rightarrow \ldots \rightarrow e_{n-1} \rightarrow e_n \end{array}$$

- A read returns the value of the most recent write
  - Reduction $e_0 \rightarrow e_1$ writes x in the register
  - Reduction $e_2 \rightarrow e_3$ writes y in the register
  - Reduction $e_4 \rightarrow e_5$ reads y from the register (not x!)
- Result of reads depends on reduction order
  - This is not lambda reduction, since for a lambda reduction the result is independent of the reduction order (confluence)

322

322

# Communication channel

- Here is an example of a FIFO channel:

x ↑send y ↑send x ↓receive y ↓receive

$e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow e_5 \rightarrow e_6 \rightarrow \ldots \rightarrow e_{n-1} \rightarrow e_n$

- The write happens before the read in the reduction order
  - Reduction $e_0 \rightarrow e_1$ sends x on the channel
  - Reduction $e_2 \rightarrow e_3$ sends y on the channel
  - Reduction $e_4 \rightarrow e_5$ receives x from the channel
  - Reduction $e_5 \rightarrow e_6$ receives y from the channel

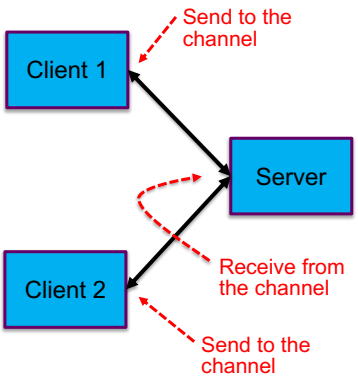
Nondeterministic!

- Order of received values depends on reduction order
  - Sending of x and y might be reversed if they are concurrent!

323

323

# Client/server application




- Let's use the communication channel to build a client/server application
  - To satisfy client liveness, the server must accept each incoming client request in a reasonable time that depends only on the travel time from the client to the server
- However, the order of the requests cannot be determined in advance because it depends on precise client timing (different timings give different reduction orders)
  - This means that the communication channel is nondeterministic
- The whole client/server application is therefore nondeterministic, even if all the other code is purely declarative

324

324

# The two most important nondeclarative operations

- Two most important nondeclarative operations are mutable state and communication channels
  - In the course we will show how to use both of them
- Mutable state: called cells
  - Leads to shared-state concurrency (Java)
  - Locks, monitors, transactions
- Communication channels: called ports
  - Leads to message-passing concurrency (Erlang)
  - Multi-agent programming

325



# Two definitions of declarative programming

326

# Two definitions

- You will notice that we have made two definitions of declarative programming
  - "A program is declarative if for all possible inputs, all executions either do not terminate or they terminate and give logically equivalent results" (lecture 3)
    - This is an observational definition: we observe a program from the outside, we don't care how the program is implemented
  - "A program is declarative if it is equivalent to an execution of a program in lambda calculus" (lecture 5)
    - This is a structural definition: it is based on how the program is implemented

327



# Comparing the definitions

- Observational is strictly more general than structural
  - All lambda executions are declarative when observed from the outside
  - An observational declarative program can be implemented using mutable state, as long as the state has no observable effect (it is hidden)
- The observational definition is best for designing programs
  - It captures the idea that the program must be deterministic even if it is concurrent, just as with lambda calculus (Church-Rosser theorem)
  - We can use mutable state in the implementation, as long as it is hidden
    - This is important because mutable state is a fundamental part of today's processors, so the low-level parts of the implementation must use it!
- The structural definition gives the theoretical basis
  - It shows that declarative programming is possible and practical
    - The Church-Rosser theorem is an important and nonobvious result!

328

# Conclusions





# Conclusions

- Declarative paradigms are the best but they cannot always be used
  - We investigate their limitations and how to overcome them
- Declarative paradigms are based on lambda calculus, which makes them confluent but they cannot interact with the real world
- To interact with the real world, we extend declarative paradigms with imperative concepts, like mutable state or communication channels
  - Mutable state (cells) leads to shared-state concurrency (Java)
  - Communication channels (ports) lead to message-passing concurrency (Erlang)
- Programs should use declarative paradigms as much as possible with as few imperative concepts as possible
  - The extensions should only be used in special cases, namely where interaction with the real world is needed