MI-GLSD-M1 -UEM213 : Programming Paradigms

Chapter VI: Concurrent Paradigm

A. HARICHE
University of Djilali Bounaama, Khemis Meliana (UDBKM)
Faculty of sciences & technology
Mathematics & computer sciences department

a.hariche@univ-dbkm.dz

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)
  - **Operating system:** management of a single computer
    - A concurrent activity is called a process
    - Processes 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
    - Communicate: information passes from one activity to another
    - Synchronize: an activity waits for another to perform a specific action

Concurrency can be (very) hard

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

Deterministic dataflow

- There are three main paradigms of concurrent programming
- The simplest is called deterministic dataflow
  - That is what we are going to see now
  - It supports all the techniques of functional programming

- What are the two other paradigms?
  - Message-passing concurrency (e.g., Erlang and Scala actors)
    - Activities send messages to each other (like sending letters)
    - Relatively straightforward, can be combined with dataflow
  - Shared-state concurrency (e.g., Java monitors)
    - Activities share the same data and they try to work together without getting in each other's way
    - Much more complicated
    - Unfortunately, many current languages still use this paradigm

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?

HARICHE A. a.hariche@univ-dbkm.dz

What to do with an uninitialized variable?

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

HARICHE A. a.hariche@univ-dbkm.dz

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?

Continuing the execution

- The waiting instruction:

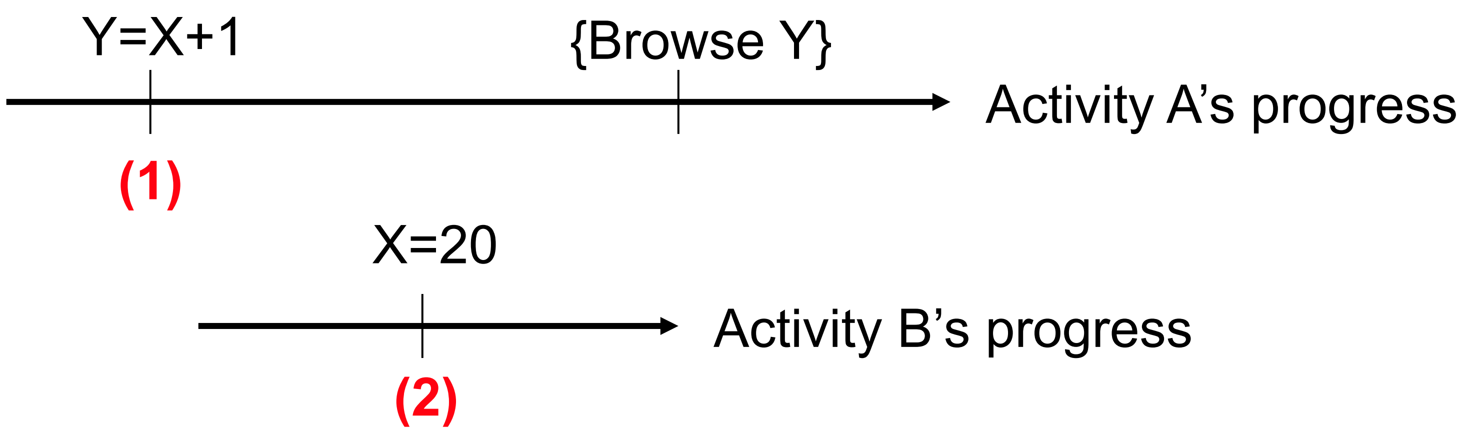
```
declare X
local Y in
    Y=X+1
    {Browse Y}
end
```

- 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

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

Threads

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

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!

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

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)

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

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 X2 X3]

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

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]

The Browser is a dataflow program

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

Streams

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

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!

Producer/ consumer (1)

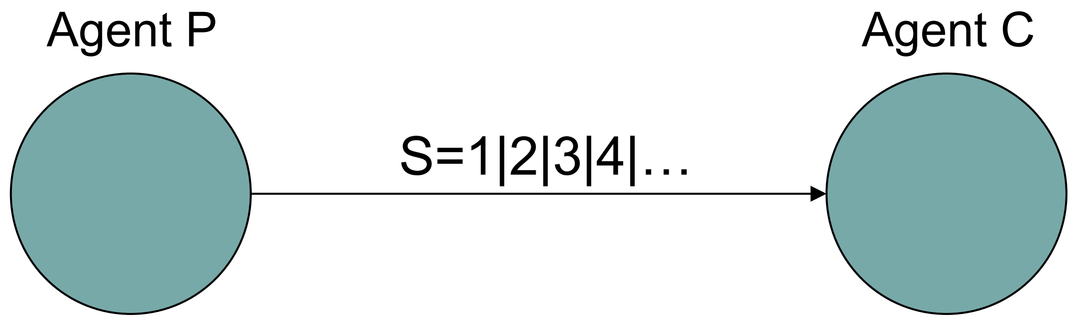
- A producer generates a stream of data

```
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
```

Producer/ consumer (2)



thread S={Prod 1} **end**

thread {Disp S} **end**

- 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

Pipeline (1)

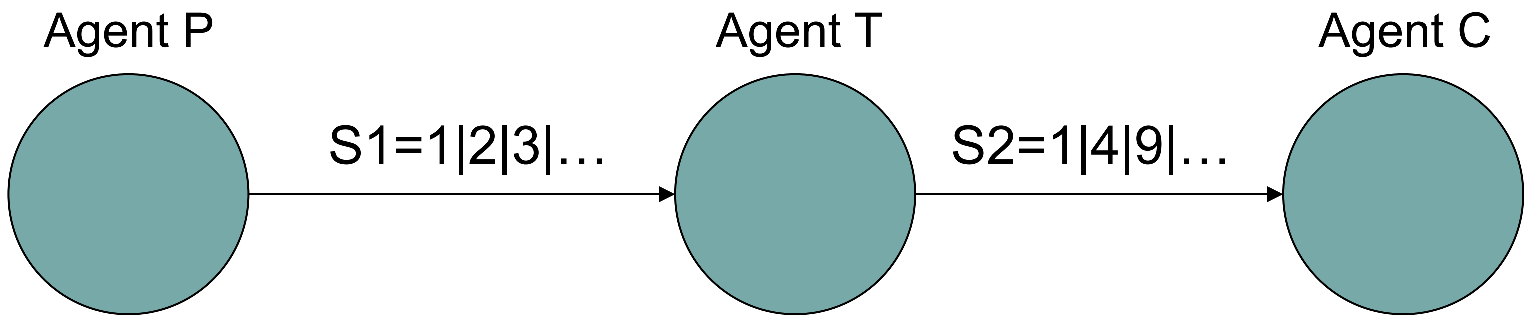
- We can add more agents between P and C
- Here is a transformer that modifies the stream:

```
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
```

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

Agents

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

Deterministic concurrency

- Each thread in a deterministic dataflow program always executes the same instructions in the same order
  - This is true even though the threads can vary their relative speeds from one execution to the next
  - Speeds can vary because of input/output, hardware interrupts, cache misses, and other sources of timing changes

- A deterministic dataflow program always gives the same outputs for the same inputs, despite variations in thread speeds
  - We say the program has no observable nondeterminism (no race conditions)
  - This is a major advantage of the deterministic dataflow paradigm that is not shared by the two other paradigms

Nondeterminism and the scheduler

- Nondeterminism is the ability of the system to make decisions that are visible by a running program
  - The application programmer does not make the decisions
  - The decisions can vary from one execution to the next
- The scheduler is the part of the system that decides at each moment which thread to execute
  - This decision is called nondeterminism
- Nondeterminism is a property of any concurrent system
  - It must be, since the concurrent activities are independent
  - A crucial part of any concurrent program is how to manage its nondeterminism

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

Example of nondeterminism (2)

- What does the following program do?

```
declare X={NewCell 0}
thread X:=1 end
thread X:=2 end
```

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

Example of nondeterminism (3)

- What does the following program do?

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

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

Managing nondeterminism

- Nondeterminism *must* always be managed
  - It should not affect program correctness
  - The most complicated case is when threads and cells are used in the same program (see previous example)
  - Unfortunately, this is exactly how many languages handle concurrency

- Deterministic dataflow has a major advantage
  - The result of a program is always the same (*except* if there is a programming error – if a thread raises an exception)
  - The nondeterminism of the scheduler does not affect the result

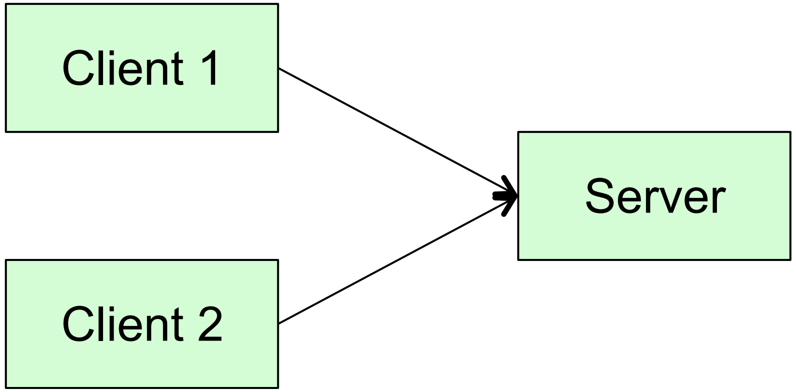
Deterministic dataflow summary

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

Concurrency *must* get simpler

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

But is determinism the right default?



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

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

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

History of deterministic dataflow

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

Next lesson

- General programming techniques for deterministic dataflow
  - « Concurrency for dummies »
- More sophisticated programming with deterministic dataflow
  - Higher-order programming and concurrent deployment
- Semantics of threads: how concurrency extends the abstract machine
  - A small extension to our abstract machine

Deterministic dataflow techniques and semantics

- Concurrency transparency
  - Adding threads to make a program more incremental, without changing the result
- A **for** loop abstraction that collects results
  - Using cells to build concurrency abstractions
- Multi-agent programming
  - Sieve of Eratosthenes: dynamically building a pipeline of concurrent agents
  - Digital logic simulation: using higher-order programming together with deterministic dataflow
- Thread semantics
  - Extending the abstract machine with multiple semantic stacks

Concurrency transparency

- We saw that multi-agent programs are deterministic
  - Their nondeterminism is not observable
  - The agent Trans with input 1|2|3|_ always outputs 1|4|9|_
- In these programs, concurrency does not change the result but only the order in which computations are done (that is, when the result is calculated)
  - It is possible to add threads at will to a program without changing the result (we call this concurrency transparency)
  - The only effect of added threads is to make the program more incremental (to remove roadblocks)
- Concurrency transparency is only true of declarative paradigms
  - It is no longer true when using cells and threads together (Java!)

Example of transparency (1)

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

Example of transparency (2)

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

Example of transparency (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

HARICHE A. a.hariche@univ-dbkm.dz

Example of transparency (4)

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

- What happens when we execute:

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

Example of transparency (5)

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

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

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

Example of transparency (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 add:
  F = fun {$ X} X+1 end

Example of transparency (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]

“Concurrency for dummies”

- Threads can be added at will to a functional program without changing the result
- Therefore it is very easy to take a functional program and make it concurrent
- It suffices to insert **thread** … **end** in those places that need concurrency

- Warning: concurrency for dummies does not work in a program with explicit state (= with cells)!
  - For example, it does not work in Java
  - En Java, concurrency is handled with the concept of a monitor, which coordinates how multiple threads access an object. This is *much more complicated* than deterministic dataflow.

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
```

Why does it work? (2)

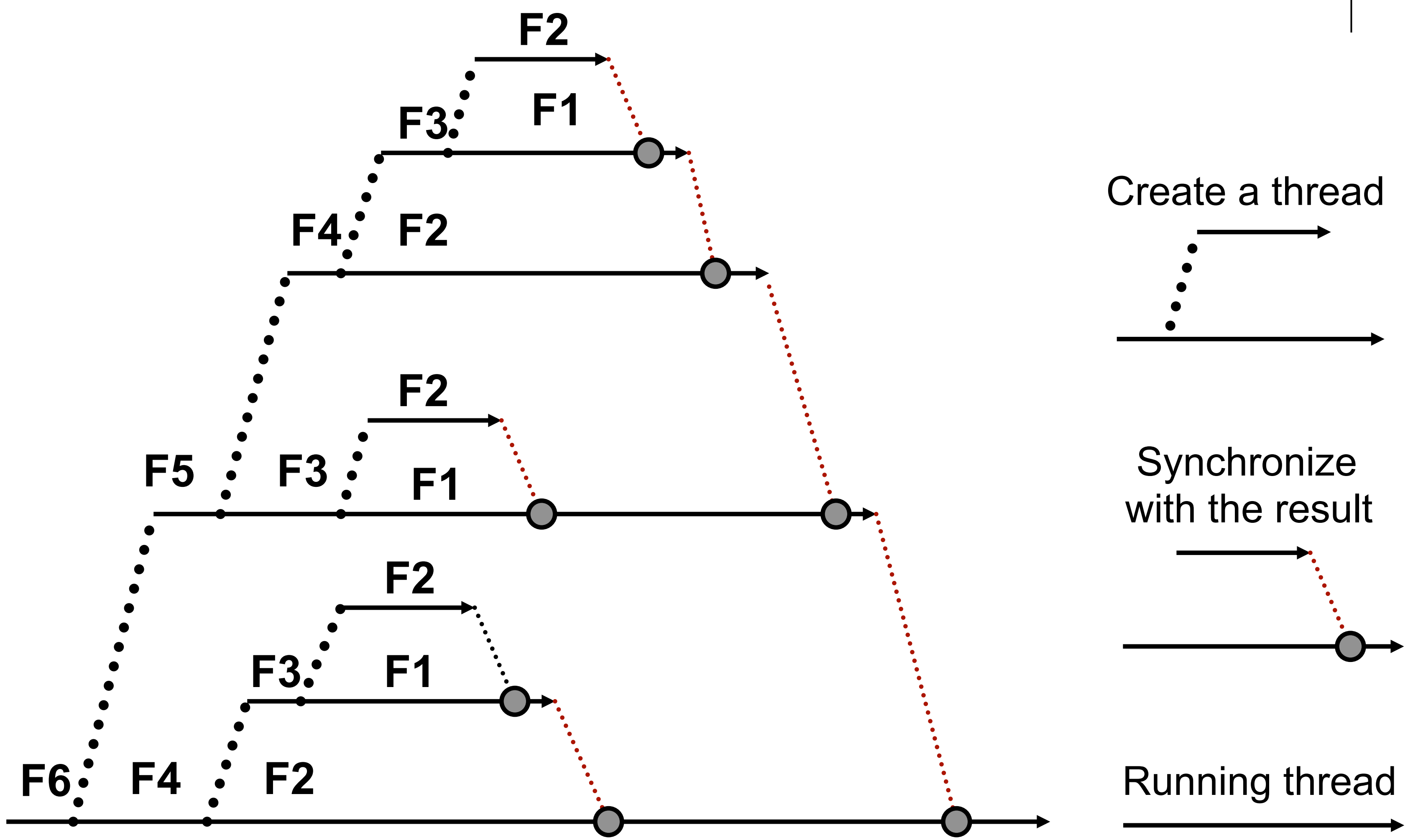
```
fun {Fib X}
   if X==0 then 0 elseif X==1 then 1
   else F1 F2 in
      F1 =  thread {Fib X-1} end
      F2  = {Fib X-2}

      F1 + F2
   end
end
```

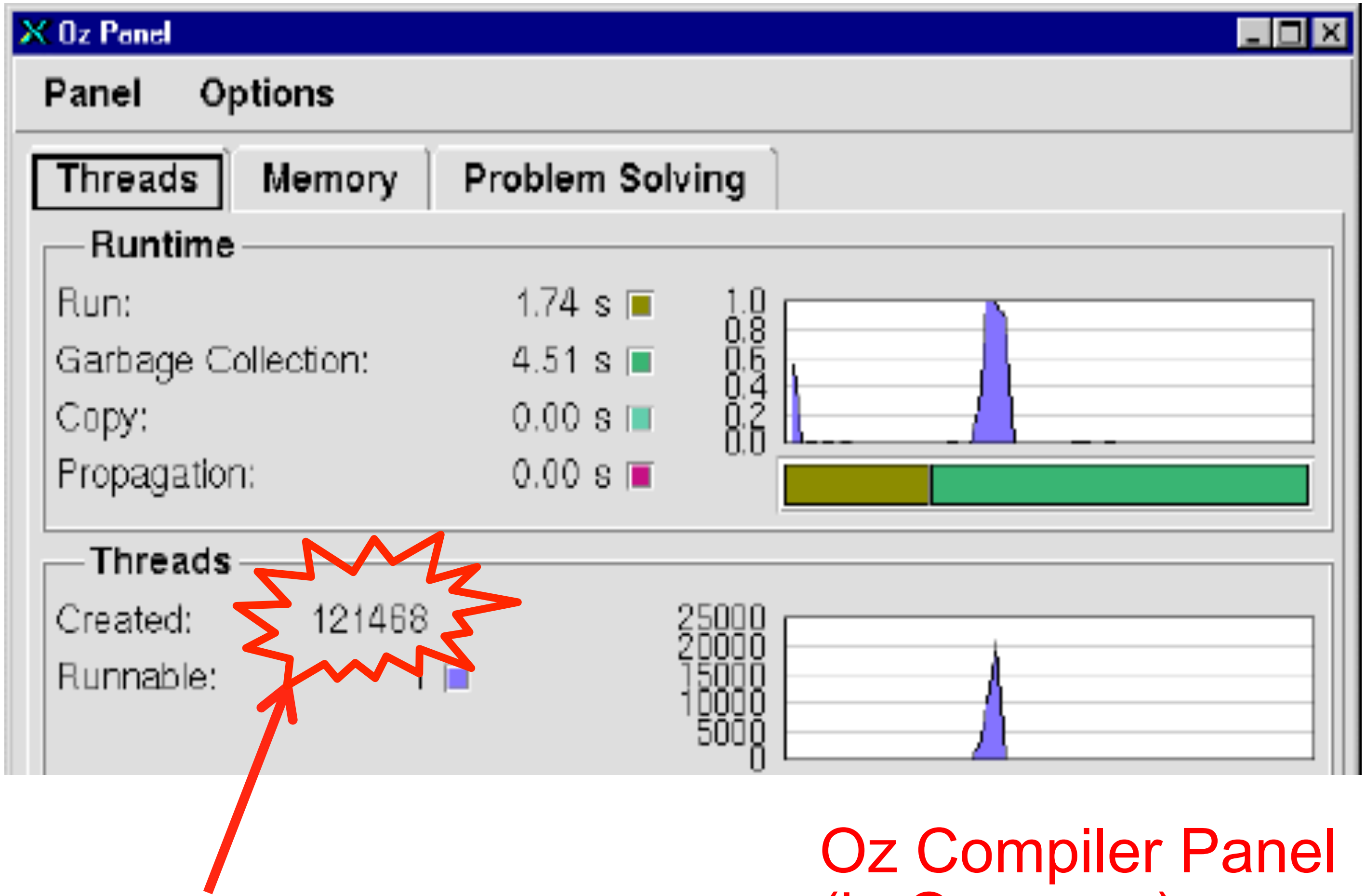
Dataflow dependency

It works because variables can only be bound to one value

Execution of {Fib 6}



Observing the execution of Fib



Total number of threads
created since system startup

Oz Compiler Panel
(in Oz menu)

A **for** loop abstraction that collects results

- We show how to use state (a cell) and higher-order programming *together* to build a powerful new abstraction for deterministic dataflow
  - The imperative and functional paradigms are not antagonistic! Using cells can give extra power to dataflow programs.

- Our new abstraction will generalize the declarative **for** loop of Oz to collect results
  - It is a powerful form of list comprehension

Declarative for loop

- Oz has a declarative **for** loop

```
for I in [1 2 3] do {Browse I*I} end
```

- This is exactly the same as executing the following three statements one after the other:

```
local I=1 in {Browse I*I} end
local I=2 in {Browse I*I} end
local I=3 in {Browse I*I} end
```

- Each iteration is independent; the identifier I references one element of the list in each iteration

Collecting results in the for loop

- We would like to extend the declarative **for** loop to accumulate results

  R = **for** I **in** [1 2 3] **do** *(accumulate I*I)* **end**

- We would like this to return R=[1 4 9]
- The existing **for** loop cannot do this, but we will define a new abstraction that can

The ForCollect abstraction

- The ForCollect abstraction extends the **for** loop with the ability to accumulate results:

  R = {ForCollect [1 2 3] **proc** {$ C I} <stmt> **end**}

- The loop body is <stmt>
  - I is the loop index
  - C is the « collect procedure »: calling {C X} in the loop body will accumulate X in R

  R = {ForCollect [1 2 3] **proc** {$ C I} {C I*I} **end**}
  ⇒ R=[1 4 9]

Defining the collect procedure (1)

- How can we define the collect procedure C?
  - C cannot be written in the functional paradigm because it has memory: each time we call {C X} we need to append X to the output list. Each time we call C the output changes.
- C can only be defined using state, i.e., a cell
  - The cell is used to append X to the output list
- But seen from the outside, ForCollect will still be functional!
  - Let us see how to define the collect procedure...

Defining the collect procedure (2)

- Assume we are building the output list and we have already added three elements to it:

  R = 1|4|9|R1

- To add another element, we need to bind R1:

  R1=16|R2

- This makes the new R = 1|4|9|16|R2
  - The new end of this list is R2!
  - So the cell always has to store the end of the list

Defining the collect procedure (3)

- We can define the collect procedure like this:

```
Acc={NewCell R} % Cell Acc contains end of the list

proc {C X}
  R2                % New end of list
in
  @Acc=X|R2         % Bind old end of list to X|R2
  Acc:=R2           % Set C to new end of list R2
end
```

- This appends X to the output list

Definition of ForCollect

- This gives us the following definition of ForCollect:

```
proc {ForCollect Xs P Ys}
   Acc={NewCell Ys}
   proc {C X} R2 in @Acc=X|R2 Acc:=R2 end
in
   for X in Xs do {P C X} end
   @Acc=nil
end
```

Doing Acc:=nil would be wrong! Do you see why?

- We need to write ForCollect as a procedure, even though we will call it as a function
  - It is because we need to access the output Ys (= initial content of Acc)

Concurrent agent with ForCollect

- We have defined ForCollect on lists, but it can do more!
  - ForCollect also works on streams

- Running ForCollect in a thread makes a concurrent agent:

```
Ys=thread {ForCollect Xs
              proc {$ C X} if X mod 2 == 0 then {C X*X} end end}
    end
```

- This agent reads an input stream Xs and returns an output stream Ys that contains the squares of the even elements of Xs

Conclusions of ForCollect

- ForCollect is a powerful abstraction that combines and generalizes both Map and Filter
  - When used with lists, it is called a list comprehension
  - Some languages have syntax for this, e.g., Haskell and Python
  - In Oz, list comprehensions can be concurrent agents
- ForCollect is defined by combining cells and higher-order programming
  - There is no antagonism between the imperative and functional paradigms; they can be used together to the benefit of both
  - Even though ForCollect uses a cell internally, it is completely deterministic when viewed from the outside. This is because we use the cell in a single thread.

Alternative definition of ForCollect

- If the collect procedure C might be used in more than one thread, then we need to change its definition to use Exchange:

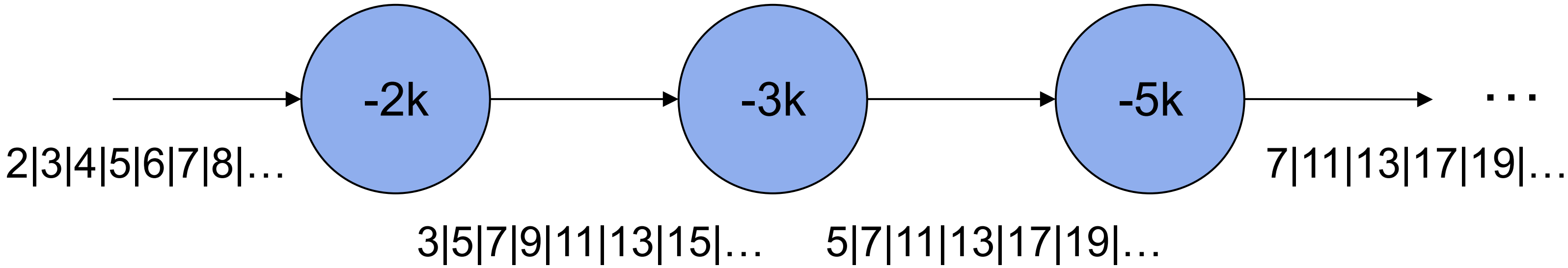
```
proc {ForCollect Xs P Ys}
   Acc={NewCell Ys}
   proc {C X} R2 in {Exchange Acc X|R2 R2} end
in
   for X in Xs do {P C X} end
   {Exchange Acc nil _}
end
```

- {Exchange Acc Old New} does two operations atomically:
  - Old is bound to the old content and New becomes the new content
  - This avoids errors when cells are used by multiple threads: doing @Acc and Acc:=R2 as two separate operations would permit another operation on Acc to be done in between, which is wrong!

Multi-agent programming

- In the last lesson 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

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
```

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
```

Concurrent deployment: building the infrastructure of a program during execution

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

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
```

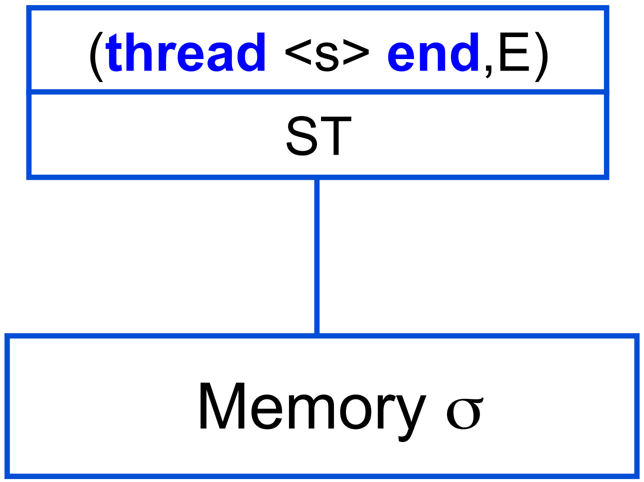
- We call {Sieve2 Xs 316} to generate a list of primes up to 100000 (why?)

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
  - This is called interleaving semantics

Thread semantics (2)

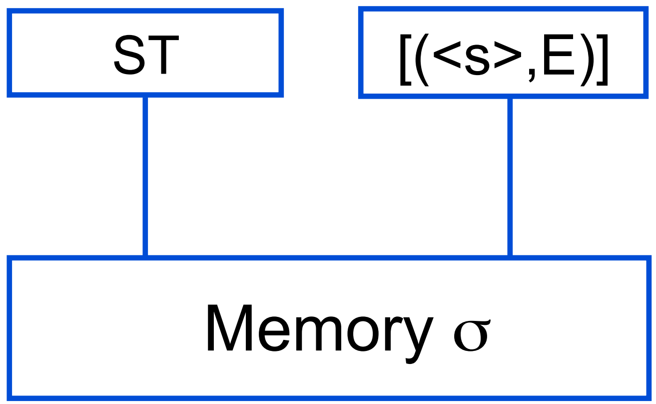
A semantic stack that is about to create a thread



HARICHE A. a.hariche@univ-dbkm.dz

Thread semantics (3)

We now have two stacks!



Why interleaving semantics?

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

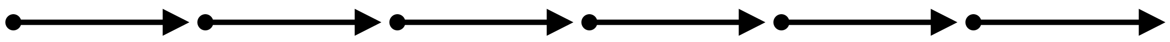
- Interleaving semantics is much easier to reason about than truly concurrent semantics
  - Truly concurrent semantics also models the case where threads ”step on each others’ toes”, but usually this is not needed, since the hardware is careful to keep this from happening
  - For example, in a multicore processor the cache coherence protocol avoids simultaneous operations on one memory word

Order of execution states

- In a sequential program, execution states are in a total order
  - Total order = when comparing any two execution states, one must happen before the other
- In a concurrent program, execution states of the same thread are in a total order
  - The execution states of the complete program (with multiple threads) are in a partial order
  - Partial order = when comparing any two execution states, there might be no order between them (either may happen first)
- In a concurrent program, many executions are compatible with the partial order
  - In the actual execution, the scheduler chooses one

Total order in a sequential program

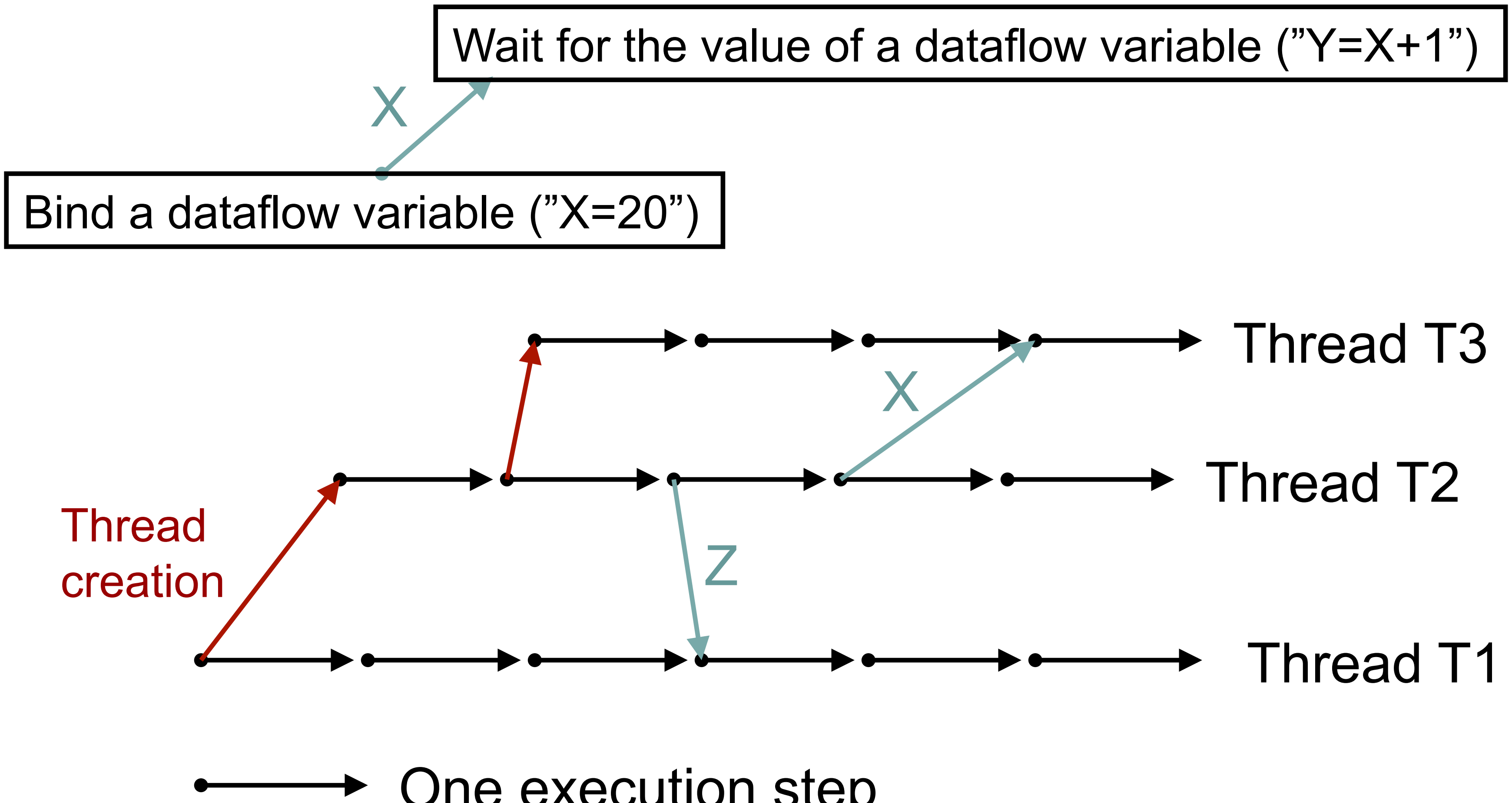
- In a sequential program, execution states are in a total order
- A sequential program has one thread
- Earlier paradigms always had this situation



Sequential execution

One execution step

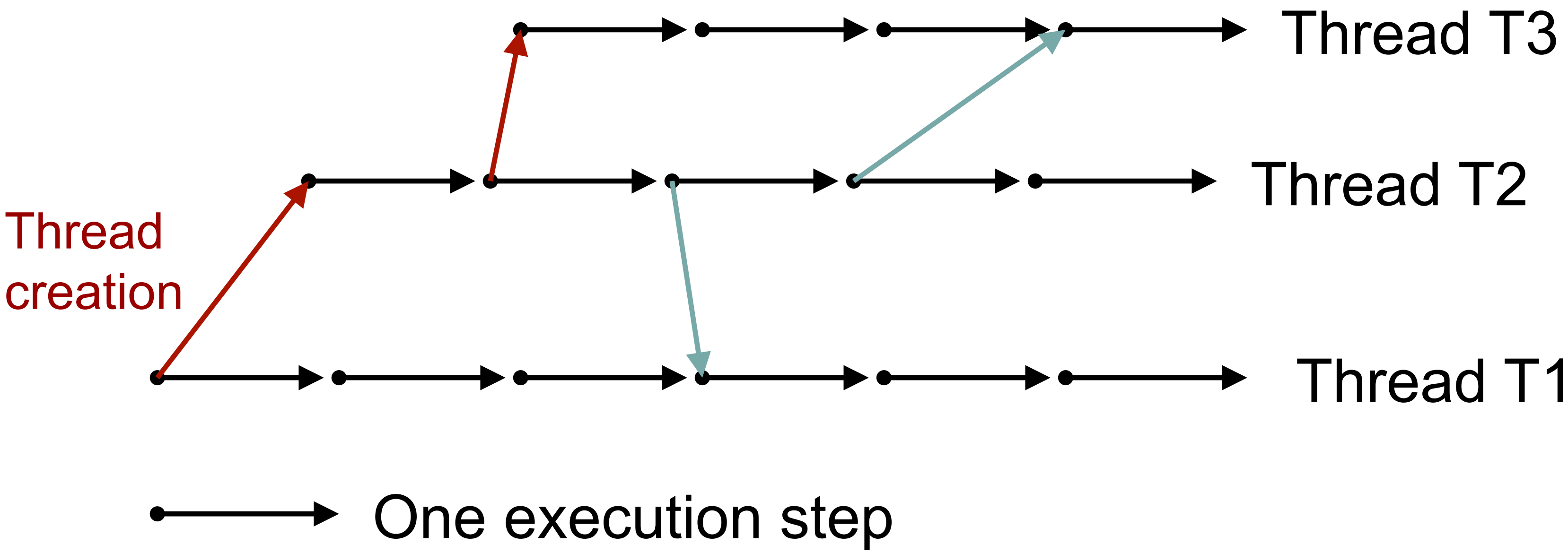
Partial order in a concurrent program



HARICHE A. a.hariche@univ-dbkm.dz

Partial order in a concurrent program

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



Digital logic simulation

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

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

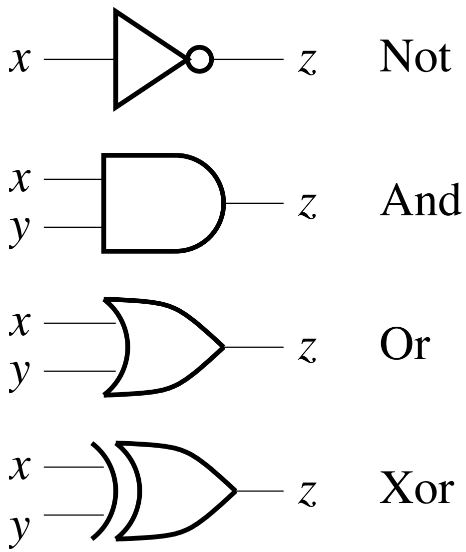
Digital signals as streams

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

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

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

Digital logic gates



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

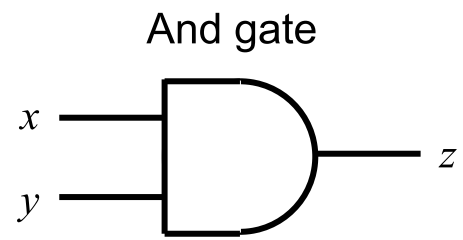
- Some typical logic gates with their standard pictorial symbols and the boolean functions that define them
- But gates are not just boolean functions!

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

Creating many gates

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

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
```

Making gates

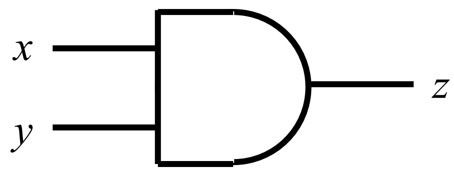
- Each of these functions can make gates:

```
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}
```

Combinational logic

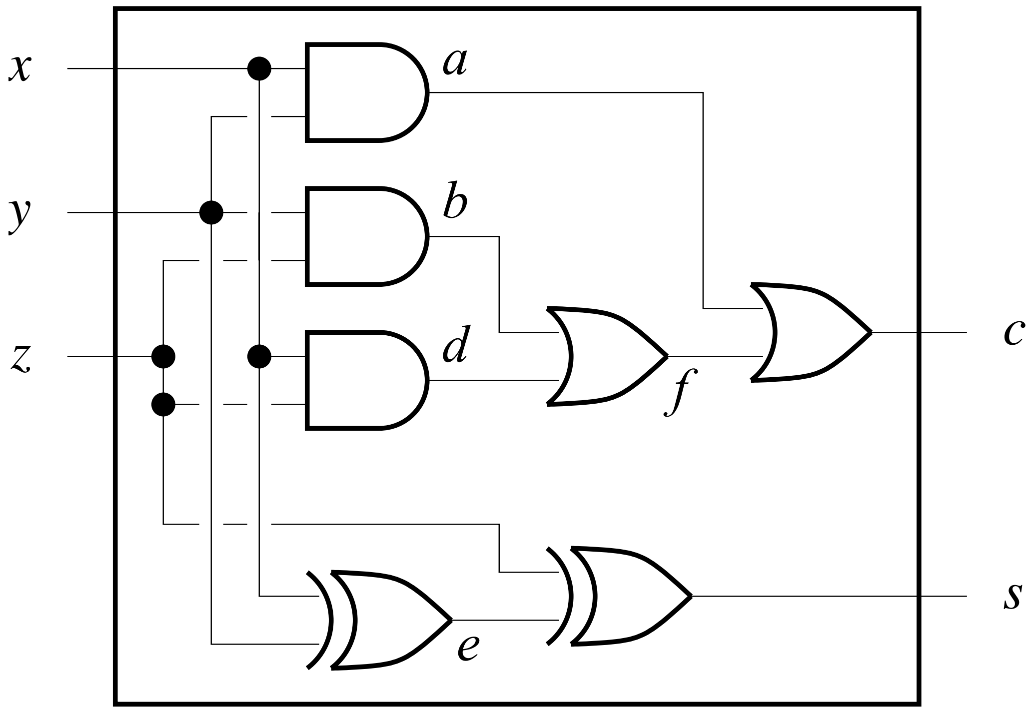
- Combinational logic has no memory: all calculation is done at the same time instant
- A gate is a simple combinational function:

$z_i = x_i$ And y_i



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

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

Full adder implementation

- Full adder creation as five-argument component:

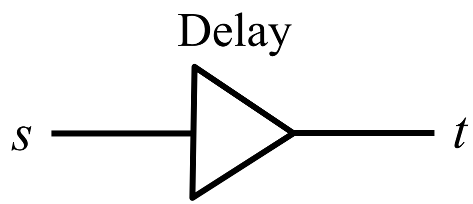
```
proc {FullAdder X Y Z C S}
   A B D E F
in
   A={AndG X Y}
   B={AndG Y Z}
   D={AndG X Z}
   F={OrG B D}
   C={OrG A F}
   E={XorG X Y}
   S={XorG Z E}
end
```

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

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

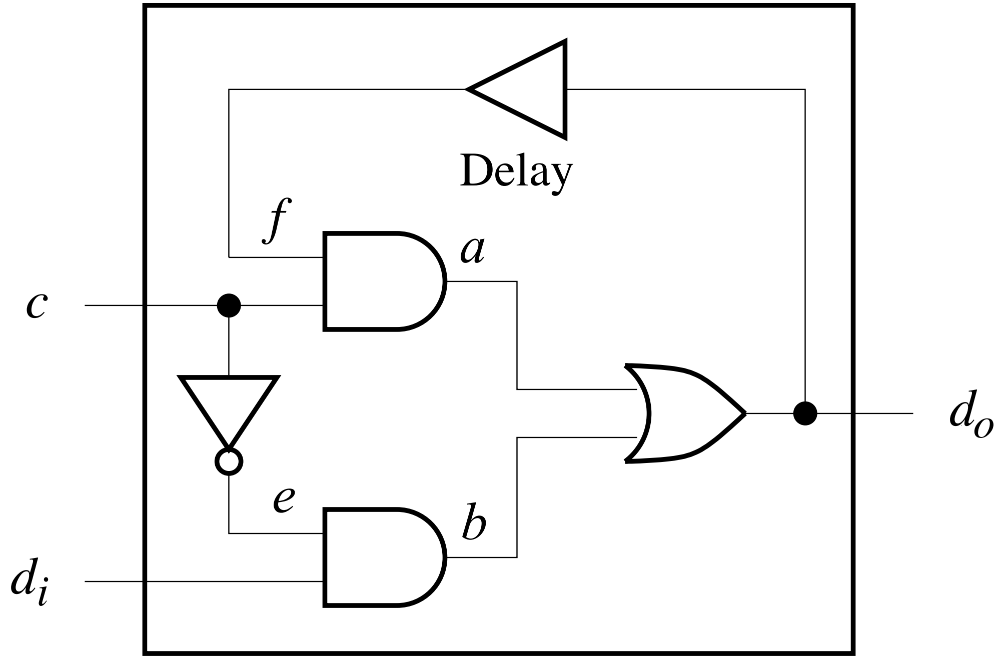
$T=b_0|b_1|b_2|\ldots|b_i|\ldots$



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

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

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

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
```

Conclusions for deterministic dataflow

- Deterministic dataflow generalizes the functional paradigm
  - There is no observable nondeterminism
  - All functional patterns become concurrency patterns
- Concurrency is transparent: « concurrency for dummies »
  - Threads can be added at will without changing the result
  - To remove roadblocks and make computation more incremental
- Deterministic dataflow is a good default
  - Nondeterminism can be added where needed and nowhere else
  - Deterministic concurrency is seeing a well-deserved resurgence after decades of neglect, at both large and small scales (big data computing and multicore computing)

Learning more about concurrency

- Paradigms that can avoid race conditions
  - Deterministic dataflow
  - Lazy deterministic dataflow
  - Constraint programming
  - Others (e.g., E: capability-based programming)
- Paradigms that can express nondeterminism
  - Message-passing concurrency
    - Scala, Erlang
  - Shared-state concurrency
    - Transactions
    - Monitors (only recommended for legacy systems)

Multi-agent dataflow paradigm

- We can combine deterministic dataflow and message passing
  - We add one concept to deterministic dataflow: a *named stream* (*port*)
  - This adds nondeterminism (*any* thread can send a message to the port)
  - Since the named stream is still a stream, it can be used in deterministic dataflow programs
- This gives multi-agent dataflow programming
  - This paradigm allows adding nondeterminism only where needed
  - Concurrency patterns can be written very concisely
    - A simple contract-net protocol can be written in just three lines
  - Ozma was an experiment to extend Scala to support multi-agent dataflow. This worked quite well, but it needs fine-grained concurrency (cheap threads) to achieve maximum usefulness (only partial success on JVM).
- Multi-agent dataflow is the best all-round concurrent paradigm
  - Even better than Erlang, since it allows managing nondeterminism

HARICHE A. a.hariche@univ-dbkm.dz

Many important ideas

Louv1.1x

- Identifiers and environments
- Functional programming
- Recursion
- Invariant programming
- Lists, trees, and records
- Symbolic programming
- Instantiation
- Genericity
- Higher-order programming
- Complexity and Big-O notation
- Moore's Law
- NP and NP-complete problems
- Kernel languages
- Abstract machines
- Mathematical semantics

Louv1.2x

- Explicit state
- Data abstraction
- Abstract data types and objects
- Polymorphism
- Inheritance
- Multiple inheritance
- Object-oriented programming
- Exception handling
- Concurrency
- Nondeterminism
- Scheduling and fairness
- Dataflow synchronization
- Deterministic dataflow
- Agents and streams
- Multi-agent programming

PLP_Drive space

https://drive.google.com/drive/folders/1YBCIZzAldeiT19DIfDiREQwP-NAQ1qMN