

Ozma: Extending Scala with Oz Concurrency



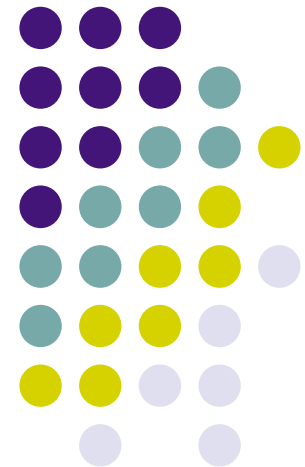
Peter Van Roy
Sébastien Doeraene

QCon 2011 San Francisco

Nov. 18, 2011

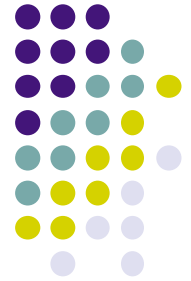
PLDC Research Group
(pldc.info.ucl.ac.be)

Université catholique de Louvain
B-1348 Louvain-la-Neuve, Belgium

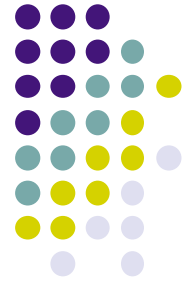


News flash:

Concurrency is still hard

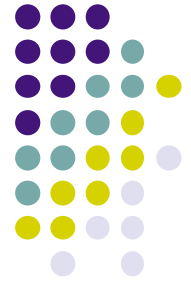


- It has all kinds of fun problems like race conditions, reentrancy, deadlocks, livelocks, fairness, scheduling, handling shared data, and multi-agent collaboration algorithms
 - Java's **synchronized objects** are tough to program with (*expletive deleted*)
 - Erlang's and Scala's **actors** are better, 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, amazingly it can be done for deterministic concurrency
 - Ozma, a conservative extension to Scala, is designed to do this



Overview of the talk

- Why deterministic concurrency?
 - Advantages and disadvantages
 - Scala + Oz $\Rightarrow$ Ozma
- Declarative dataflow
 - Lightweight threads and the wonders of single assignment `val`
- More programming techniques
 - Three powerful principles
- Message passing and nondeterminism
 - This is also very important, so let's add it cleanly
- Technical details for the language geek
 - Comparison with Scala streams and lazy vals, handling exceptions
- The past is prologue
 - The future of Ozma, distribution, and fault tolerance



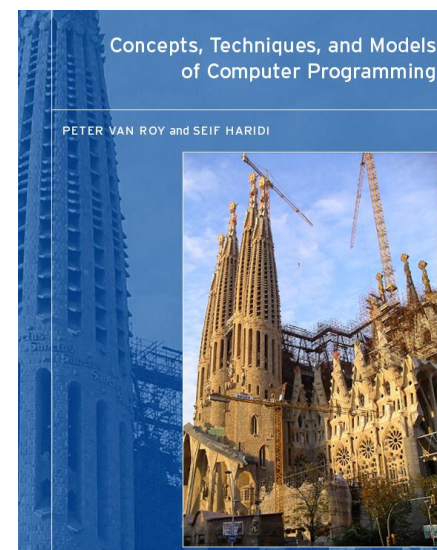
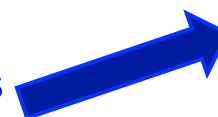
Scala in a nutshell

- Scala is a multiparadigm language that compiles to JVM and .NET.
 - Directly interoperable with Java
- Developed since 2001 by Martin Odersky and others, it supports both functional programming and object-oriented programming
 - Clean language with advanced properties (e.g., closures, powerful type inferencing), easy migration for Java programmers
- According to Typesafe, Inc., it has over 100,000 developers
 - See www.scala-lang.org for more information
- Scala provides common abstractions for concurrent programming
 - Signals and monitors (synchronized objects)
 - Futures, syncvars, asynchronous and synchronous channels, fork-join
 - Actors with mailboxes, semaphores
 - Akka library: transactional actors
- Some of these are good and some are bad
 - Good: **futures, syncvars, channels, fork-join** ⇒ *they are deterministic!*
 - Bad: monitors, semaphores (least bad: actors, transactions)

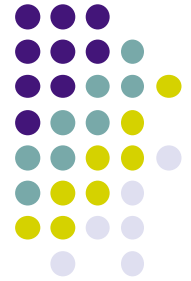


Scala + Oz $\Rightarrow$ Ozma

- Oz is a multiparadigm language that has been used for language experiments by a bunch of smart but eccentric language researchers since the early 1990s (see www.mozart-oz.org)
 - Constraint programming, network-transparent distributed programming, declarative/procedural GUI programming, concurrent programming
 - Textbook “Concepts, Techniques, and Models of Computer Programming”, MIT Press, 2004
 - Oz supports concurrent programming based on a declarative dataflow core with lightweight threads
- $\Rightarrow$ Ozma extends Scala with a new concurrency model based on the Oz dataflow ideas



One third of the book is about concurrency



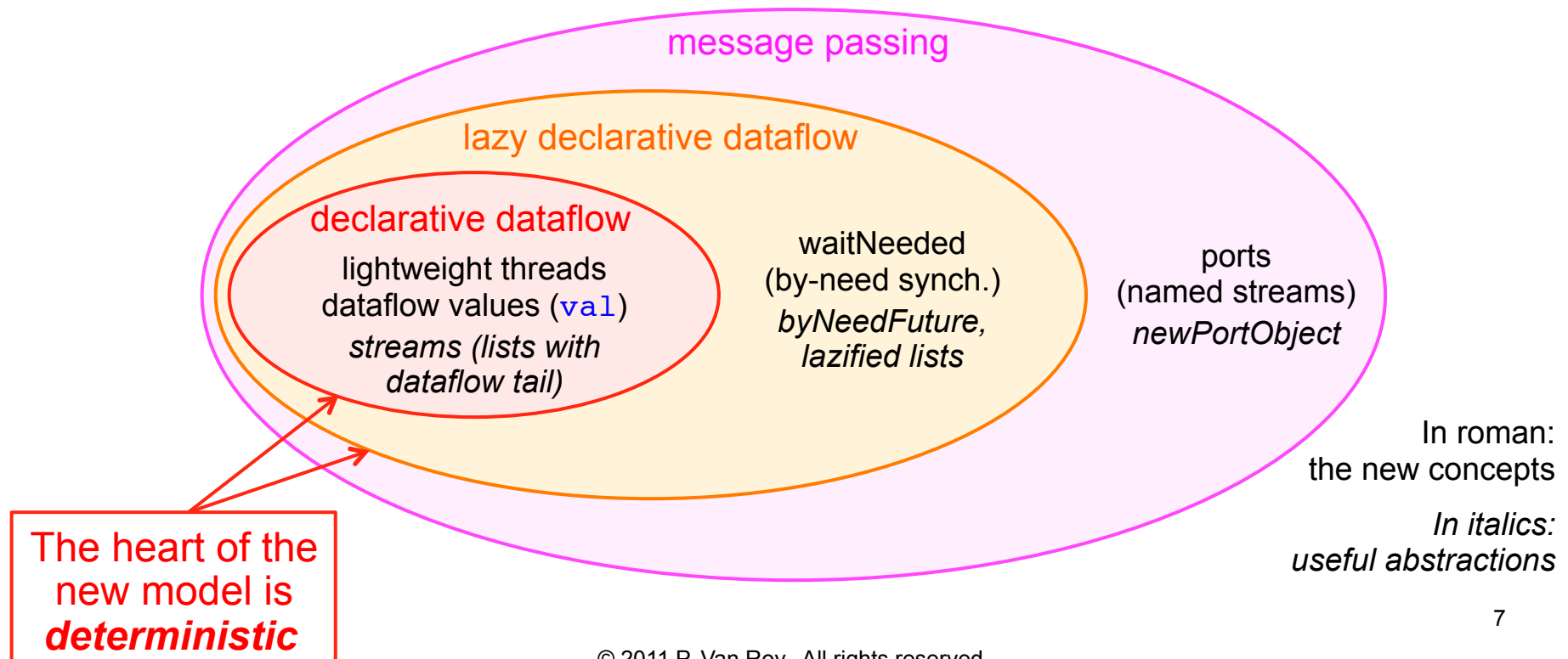
Ozma implementation

- Ozma's implementation combines a modified Scala compiler and a modified Oz compiler, and targets the Oz VM (Mozart). It was first released in June 2011.
 - The Oz VM has efficient support for lightweight threads, dataflow synchronization, by-need synchronization, and failed values
- Full source and binaries (with open-source license) available at:
<https://github.com/sjrd/ozma>
- Full documentation available at:
<http://www.info.ucl.ac.be/~pvr/MemoireSebastienDoeraene.pdf>
- Download the compiled binaries and try it out!
 - Or compile it yourself with Scala $\geq 2.9.0$, Mozart $\geq 1.4.0$, and Ant ≥ 1.6
 - It runs under Linux, Mac OS X, and maybe Windows
- All the Ozma examples in this talk are running code

Ozma extends Scala with a new concurrency model

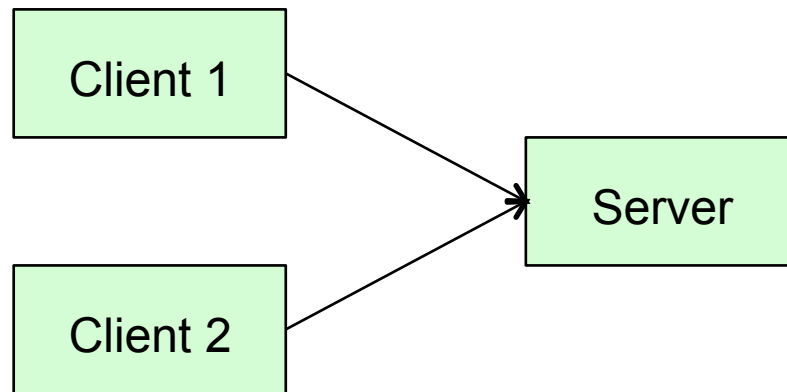


- The heart of the model is declarative dataflow
 - Further extended with laziness (still declarative) and ports (for nondeterminism)
 - This allows adding nondeterminism exactly where needed and no more





Why deterministic concurrency?



This client/server can't be written in a deterministic model!

It's because the server accepts requests nondeterministically from the two clients

- Determinism has strong limitations!
 - Any concurrent execution always gives the same results
 - 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

Deterministic concurrency: the right default?



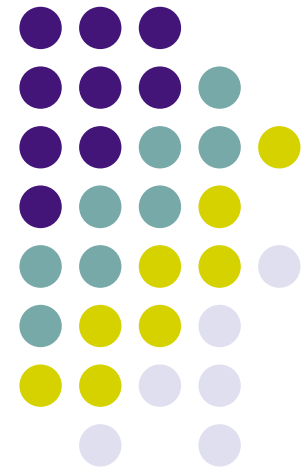
- Parallel programming has finally arrived (a surprise to old timers like me!)
 - **Multicore processors**: dual and quad today, a dozen tomorrow, a hundred in a decade, most apps will do it
 - **Distributed computing**: data-intensive with tens of nodes today (NoSQL, MapReduce), hundreds and thousands 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 concurrency is functional programming!
 - It can be extended cleanly to distributed computing
 - Open network transparency (implemented in Oz since 1999)
 - Modular fault tolerance (implemented in Oz since 2007)
 - Large-scale distribution (on the way...)

Such an old idea, why isn't it used already?

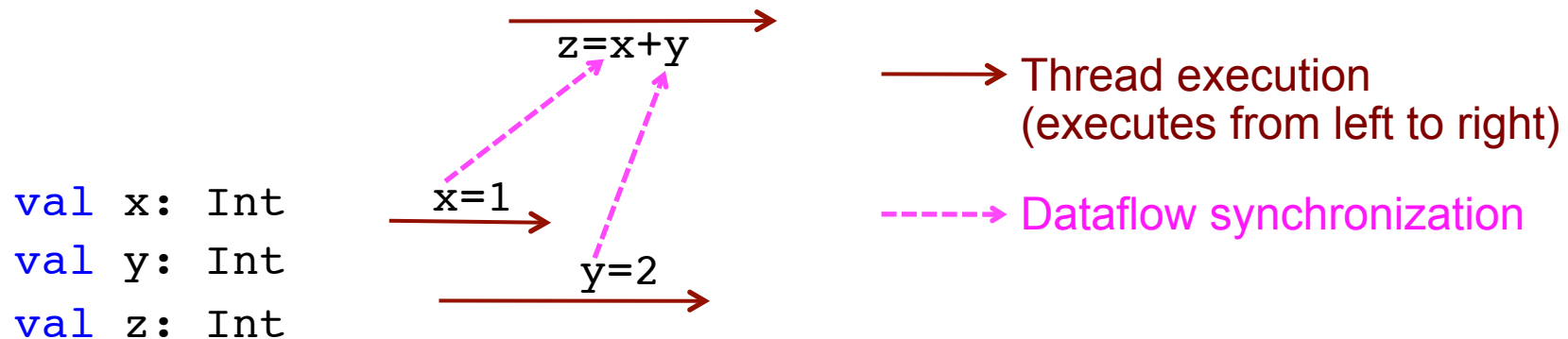


- **Deterministic concurrency** has a long history that starts in 1974
 - Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress*, pp. 471-475, 1974. *Deterministic concurrency*.
 - Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. 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 handle 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).

Declarative Dataflow



Declarative dataflow

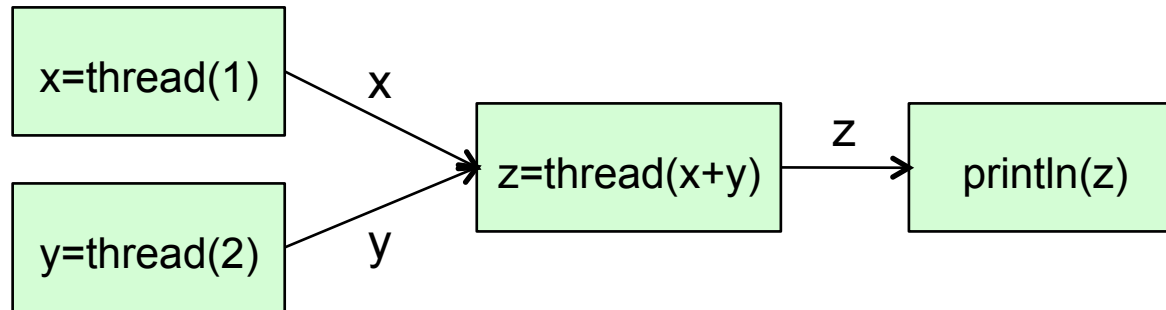


```
thread { x=1 }
thread { y=2 }
thread { z=x+y }
```

```
println(z)
```

- All `val` values can do dataflow
 - They are single assignment
 - The addition operation *waits* until both x and y are bound
 - This does both synchronization and communication
- Programs with declarative dataflow are **always deterministic**
 - This program will always print 3, independent of the scheduler

Using the thread statement as an expression



Each green box is
a concurrent agent

Each arrow is a
shared dataflow value

```
val x = thread(1)
val y = thread(2)
val z = thread(x+y)

println(z)
```

- Exactly the same behavior as the previous example
- Using the thread statement in this way can often simplify the syntax of concurrent programs

Declarative dataflow extensions to Scala



- **Lightweight threads**: hundreds of thousands of threads can be active simultaneously (like Erlang, by the way)

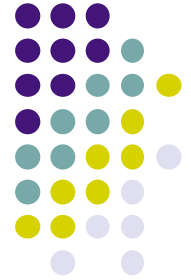
```
thread { println("New lightweight thread") }
```

- **Dataflow values**: every `val` can be a single-assignment variable. Operations that need the value will wait until it is available.

```
val x = thread(1)    // binds x in its own thread
println(x+10)        // the addition waits for x
```

- **By-need (lazy) execution**: wait until value is *needed*

```
val x: Int
thread{ waitNeeded(x); x=factorial(69) }
println(x)           // need to print causes calculation of x
```



Implementing futures

```
import scala.ozma.thread
object SimpleFuture {
  def main(args: Array[String]) {
    println("start")
    val result = thread(fibonacci(40)) // create a future
    println("continue execution while computing")
    println("Fib(40) = "+result)       // wait for result
  }

  def fibonacci(arg:Int): Int = arg match { // burn cycles
    case 0 | 1 => 1
    case _ => fibonacci(arg-1)+fibonacci(arg-2)
  }
}
```

- Futures can be implemented easily using dataflow values
- The computation is started in a new thread and returns a future
- Any calculation using the future will wait until its value is available

Streams: lists as dataflow communication channels



```
val x: List[Int]
val ints = 1 :: 2 :: 3 :: 4 :: x // unbound tail

thread { ints foreach println } // a printing agent

val y: List[Int]
x = 5 :: 6 :: 7 :: y // the agent will print these
```

- A stream is a **list with an unbound dataflow tail**
 - It can be extended indefinitely or terminated with `Nil`
- Any list function can read a stream (it's exactly like reading a list)
 - It will automatically wait when it finds an unbound tail
 - Like the `foreach` operation in this example
 - If put inside a thread, the list function becomes a **concurrent agent**

The magic of declarative dataflow

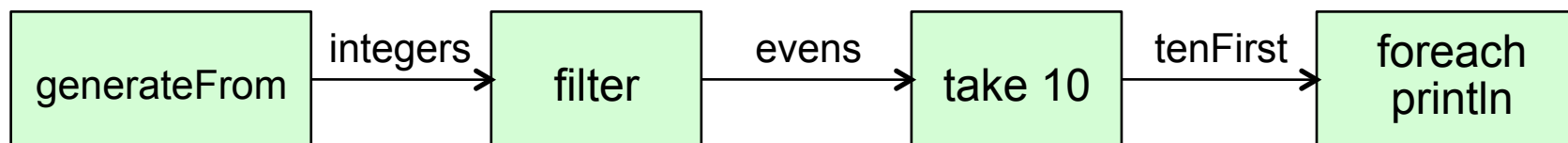


```
object Test {  
  def main(args: Array[String]) {  
    val range = gen(1, 10) // sequential version  
    val result = range map (x => x*x)  
    result foreach println  
  
    val range2 = thread(gen(1, 10)) // concurrent version  
    val result2 = thread(range map (x => x*x))  
    result2 foreach println  
  }  
  def gen(from: Int, to: Int): List[Int] = {  
    sleep(1000)  
    if (from > to) Nil  
    else from :: gen(from+1, to) // tail-recursive in Ozma  
  }  
}
```

- Both versions print the same final result 1, 4, 9, 16, ..., 100
 - So what's the difference? What does concurrency buy you?
- The sequential version: nothing is output for 10 seconds, and then the whole list
- The concurrent version: a new result is output every second
- Declarative dataflow turns batch programs into incremental programs



Pipelines using streams



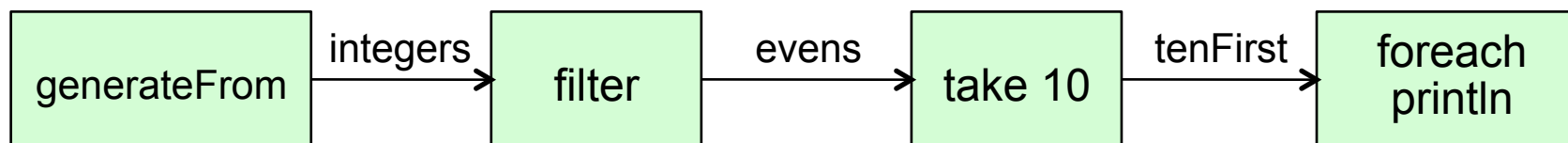
```
def generateFrom(n: Int): List[Int] =  
  n :: generateFrom(n+1)
```

```
val integers = thread(generateFrom(0))  
val evens = thread(integers filter (_ % 2 == 0))  
val tenFirst = thread(evens take 10)  
tenFirst foreach println
```

- A list function put in a thread becomes a concurrent agent
- List functions must be tail-recursive for this to work
 - This is automatically true in Ozma (ensured by compiler transformation)



Lazy pipelines

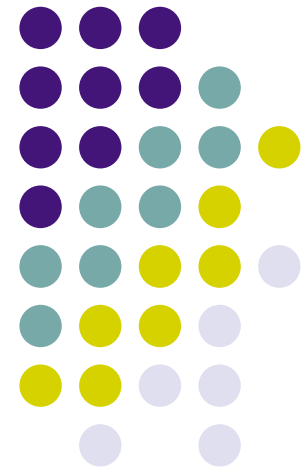


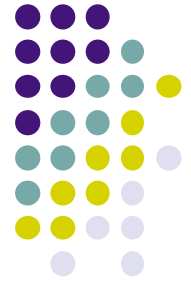
```
def generateFrom(n: Int): List[Int] = byNeedFuture {  
  n :: generateFrom(n+1)  
}
```

```
val integers = generateFrom(0)  
val evens = integers.lazified filter (_ % 2 == 0)  
val tenFirst = evens.lazified take 10  
tenFirst foreach println
```

- `byNeedFuture` introduces lazy execution: its body will be executed on demand; list operations are made lazy by modifying lists with `.lazified`
- Lazy execution preserves determinism

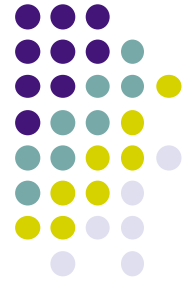
More Programming Techniques





Three powerful principles

- Any functional program can be made concurrent without changing the result by adding calls to `thread`
 - Threads can be added anywhere in the program
 - Turns batch into incremental (removes roadblocks)
- Any list function can become a concurrent agent by executing it in a thread
 - Because list functions in Ozma are tail-recursive, the agent has no memory leak (stack size is constant)
- Any computation, functional or not, can be made lazy by adding calls to `waitNeeded`
 - Syntactic sugar is provided with `byNeedFuture` and `.lazified`



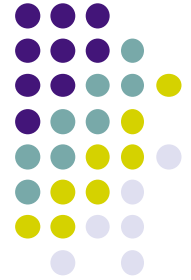
From map to concurrent map

```
def map[A, B](list: List[A], f: A => B): List[B] = {  
  if (list.isEmpty) Nil  
  else f(list.head) :: map(list.tail, f)  
}
```



```
def concMap[A, B](list: List[A], f: A => B): List[B] = {  
  if (list.isEmpty) Nil  
  else thread(f(list.head)) :: concMap(list.tail, f)  
}
```

- In concMap, all evaluations of f execute concurrently
- It is even possible to call concMap when f is not known (unbound). This will create a list containing unbound values, like futures: they will be evaluated as soon as f is known (bound to a function).



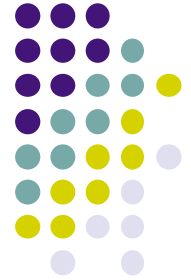
Map as a concurrent agent

```
def gen(from: Int): List[Int] = from :: gen(from+1)
```

```
def displayEvenSquares() {  
  val integers = thread(gen(0))  
  val evens = thread(integers filter (_ % 2 == 0))  
  val evenSquares = thread(evens map (x => x*x))  
  evenSquares foreach println  
}
```

Concurrent agent

- Wrapping the calls to `gen`, `filter`, and `map` within threads turns them into concurrent agents
 - Note that `foreach` is also an agent, living in the main thread
- As new elements are added to the input stream, new computed elements will appear on the output stream



Map as a lazy agent

```
def gen(from: Int): List[Int] = byNeedFuture {  
  from :: gen(from+1)  
}
```

```
def displayEvenSquares() {  
  val integers = gen(0)  
  val evens = integers.lazified filter (_ % 2 == 0)  
  val evenSquares = evens.lazified map (x => x*x)  
  evenSquares foreach println  
}
```

- Now `foreach` imposes the control flow and laziness prevents the agents from getting ahead of the consumer
- This guarantees that the whole program executes in constant memory

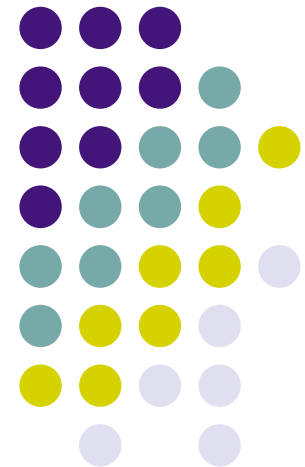
Sieve of Eratosthenes as a declarative dataflow program



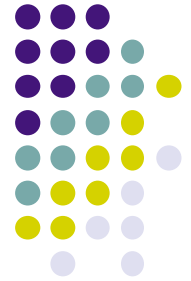
```
import scala.ozma._
object PrimeNumbers {
  def main(args: Array[String]) {
    val max = args(0).toInt
    val integers = thread(generate(2, max))
    val result = thread(sieve(integers))
    result.toAgent foreach println
  }
  def generate(from: Int, to: Int): List[Int] = {
    if (from > to) Nil else from :: generate(from + 1, to)
  }
  def sieve(list: List[Int]): List[Int] = {
    list match {
      case Nil => Nil
      case head :: tail =>
        val filtered = thread {
          tail.toAgent filter (_ % head != 0)
        }
        head :: sieve(filtered)
    }
  }
}
```

- This program calculates the prime numbers up to a maximum using the Sieve of Eratosthenes.
- The program **dynamically builds a pipeline of filter agents** that successively remove multiples of 2, 3, 5, etc.
- The program can be made lazy by prefixing the generate and sieve definitions with `byNeedFuture`

Message Passing and Nondeterminism



Managing nondeterminism with ports



- So far, all our programs have been deterministic
 - Determinism is a good default, but for real programs we need nondeterminism too!
- Let's add nondeterminism in a nice way
 - One way is to **give names to streams**
- A **port** is a **named stream**, where the name is a constant
 - Any thread can send a value to a port
 - The port will append the value to its stream
 - The senders and the receivers of a port can themselves be deterministic computations; the only nondeterminism is the order in which sent values appear on the port's stream



Introducing ports

```
val (s, p) = newPort[Int] // Create port p with stream s
thread{ p.send(1) }
thread{ p.send(2) }
thread{ p.send(3) }
```

```
thread { s foreach println } // Print elements of the
                             // port's stream one by one
```

- The values 1, 2, and 3 will be displayed in some order (nondeterminism)
 - The actual order depends on the thread scheduler
- No memory leak: garbage collection will remove the parts of the stream already read

Partial barrier synchronization with ports



```
def partialBarrier(m: Int, tasks: List[() => Unit]) {  
  val (stream, port) = Port.newPort[Unit]  
  for (task <- tasks)  
    thread { task(); port.send(()) }  
  stream(m-1) // wait for at least m elements  
}  
  
println("start")  
partialBarrier(1, List(  
  () => { sleep(1000); println("a") },  
  () => { sleep(3000); println("b") },  
  () => { sleep(2000); println("c") }  
))  
println("peekaboo")
```

- The partial barrier starts n tasks concurrently and waits until m tasks complete (with $m \leq n$)
- We implement it with a port whose stream contains only units

Building nondeterministic agents with ports



```
def newPortObject[A,B])(init:B)(handler:(B,A)=>B) = {  
  val (s, p) = Port.newPort[A]  
  thread{ s.toAgent.foldLeft(init)(handler) }  
  p  
}
```

Initial state

State updater

- A port object is an actor. It reads messages sequentially from the stream and uses the messages to update its internal state.
- The `foldLeft` operation updates the internal state as messages are received (note: s_i is a received message):

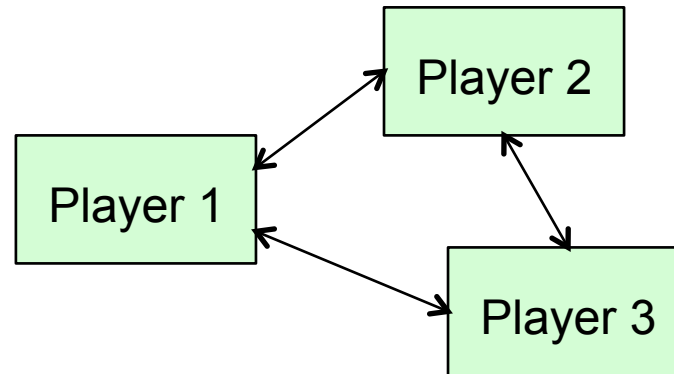
`(...(((init handler  $s_0$ ) handler  $s_1$ ) handler  $s_2$ ) ... )`

- The current value of `foldLeft` is the agent's internal state
- Neat trick: `foldLeft` is a **function used as a concurrency pattern**

Agents playing ball



```
object BallGame {
  type Ball = Unit
  val ball: Ball = ()
  type Player = Port[Ball]
  def main(args: Array[String]) {
    val player1: Player
    val player2: Player
    val player3: Player
    player1 = makePlayer("Player 1", Seq(player2, player3))
    player2 = makePlayer("Player 2", Seq(player3, player1))
    player3 = makePlayer("Player 3", Seq(player1, player2))
    player1.send(ball)
    while(true) sleep(1000)
  }
  def makePlayer(id:Any, others:Seq[Player]):
    Player = {
      Port.newPortObject(0){(st:Int, b:Ball) =>
        st+1
        Random.rand(others).send(b)
      }
    }
}
```



- Each player receives the ball and sends it to a randomly chosen other player
- Each player counts the number of balls received
- The port allows a player to receive from either of the others (nondeterminism)

Recursive thread termination (1)



- A new thread can itself create new threads, and so forth recursively
- We would like to detect when *all threads are terminated*.
This is harder than barrier synchronization since we don't know in advance how many threads are created.

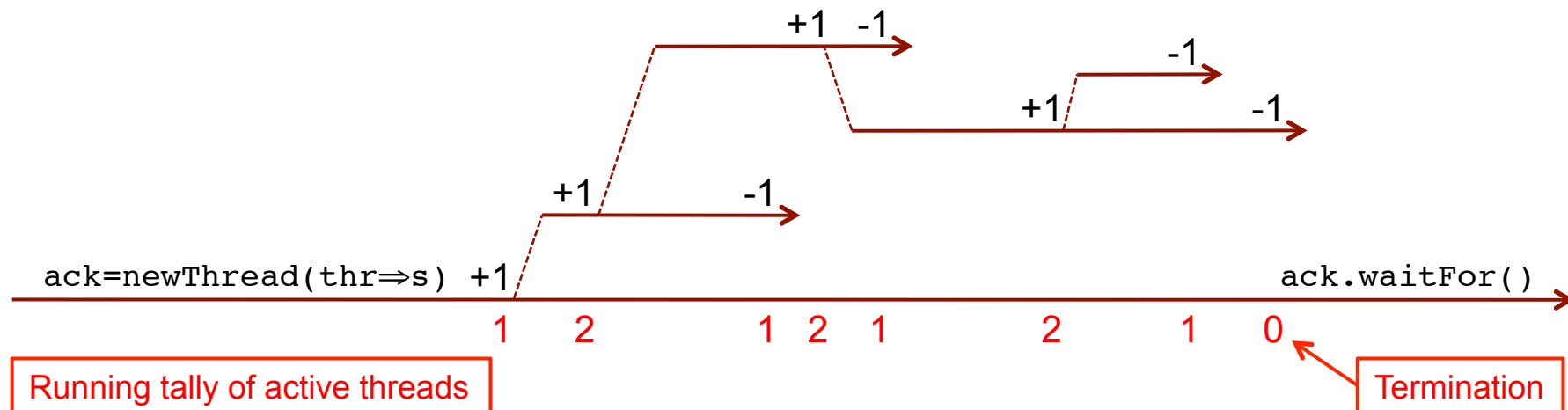
- Here's the interface:

```
newThread(SubThreadProc => Unit): Ack
```

- Here's the usage:

```
val ack = newThread { thr => s }  
          // s is any statement  
          // thr{s'} inside s creates a thread  
  
ack.waitFor() // waits until all threads have terminated
```

Recursive thread termination (2)



- How it works: we use a port to tally 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
 - Needs a bit of care to avoid races: send +1 *just before* creation and -1 *inside the thread* just before termination
 - When the running total on the stream is 0 then all threads are terminated

Recursive thread termination (3)



```
sealed class Ack {
  def waitFor() {
    // do nothing
  }
}

object Ack extends Ack

object RecursiveTermination {
  type SubThreadProc = (=> Unit) => Unit
  def newThread(body: SubThreadProc => Unit): Ack = {
    val (s, p) = Port.newPort[Int]
    def subThread(c: => Unit) {
      p.send(1)
      thread { c; p.send(-1) }
    }
    def zeroExit(n: Int, is: List[Int]): Ack = is match {
      case i :: ir => if (n+i != 0) zeroExit(n+i, ir) else Ack
      case Nil => Ack
    }
    subThread {
      body(subThread)
    }
    thread(zeroExit(0, s))
  }
}
```

Recursive thread termination (4)

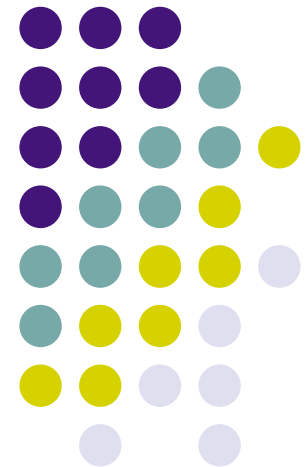


```
def main(args: Array[String]) {  
  val ack = newThread { thr =>  
    sleep(500); println("c")  
    thr { sleep(250); println("d")  
      thr { sleep(2000); println("b") }  
      thr { sleep(1000); println("a") } }  
  }  
  
  println("started")  
  ack.waitFor()  
  println("all done")  
}
```

Output
started
c
d
a
b
all done

- `newThread` creates main thread
- `thr` creates subthreads (recursive calls allowed)
- `ack.waitFor()` waits until all threads are done

Technical Details for the Language Geek



Scala streams versus Ozma streams



- **Scala streams** are lists with a delaying mechanism for the tail
 - A Scala stream provides a form of **coroutining (sequential)**
 - Lazy: The tail is calculated only when the **tail method is invoked**
- **Ozma streams** are lists with an unbound dataflow tail
 - An Ozma stream is used in **both eager and lazy concurrency**
 - Eager: Access to the tail waits until the tail is bound
 - Lazy: Calculation of the tail is initiated when the **tail is needed**
- What's the difference between Scala and Ozma streams?
 -  Ozma streams allow **slack**: the producer can get ahead of the consumer
 - It's possible to write a bounded buffer with Ozma streams, but not with Scala streams: in the latter, the producer and consumer execute in lock step
 - Ozma streams guarantee **independence**: if the producer gets in an infinite loop or raises an exception, this does not hinder the consumer

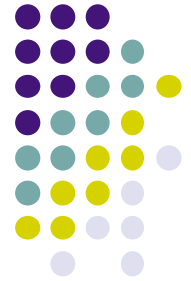
Scala lazy val versus Ozma byNeed



Scala ↔ Ozma

```
Scala { lazy val x = { ... }  
      val y = x    // x is evaluated here in Scala  
      println("checkpoint")  
      println(y)  
      }  
  
Ozma { val x = byNeed { ... }  
      val y = x    // x is not evaluated (not needed)  
      println("checkpoint")  
      println(y)  // x is evaluated here in Ozma (needed)  
      }
```

- `lazy val` is evaluated **upon first access (encounter in code)**
- `byNeed` value is evaluated **upon need (actual use)**
 - It can be passed around as an argument without evaluating it



Two features and a limitation

- Two features:
 - Errors in lazy execution are handled through **failed values**, so that the exception appears at the point where the value is needed
 - Port streams are **read-only**; they can only be extended by the port's send operation and not by any other operation (secure encapsulation)
- One limitation:
 - The current Ozma implementation has one limitation related to garbage collection and the `List` module. The methods `.lazified` and `.toAgent` must be used whenever a list operation in the standard `List` module is used, since otherwise the operation will not currently reclaim memory when used in a dataflow style. Note: for user-defined list operations the problem does not exist.
 - The limitation is due to how lists are currently implemented in Ozma; we plan to remove it in the near future.

Handling exceptions in lazy computations



```
try {  
  val list = Nil:List[Int]  
  val x = byNeedFuture(list.head)  // list is empty!  
  println(x)  
} catch {  
  case _: java.util.NoSuchElementException =>  
    println("The list was empty")  
}
```

- What happens if the lazy computation (in `byNeedFuture`) throws an exception?
 - The lazy computation is running in another thread from the thread that needs `x`!
- The only reasonable possibility is to raise the exception where `x` is needed
 - `byNeedFuture` catches the exception thrown by the lazy computation and wraps it in a **failed value**, which causes `println(x)` to raise an exception
 - The exception is raised in the right place

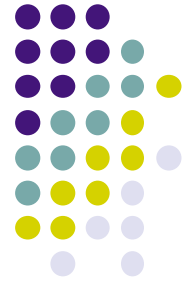
Implementation of byNeedFuture



```
def byNeedFuture[A](value: => A) = { // value is by-name
  val result: A
  thread {
    waitNeeded(result)
    try {
      result = value // value is evaluated here
    } catch {
      case throwable: Throwable =>
        result = makeFailedValue(throwable)
    }
  }
  result
}
```



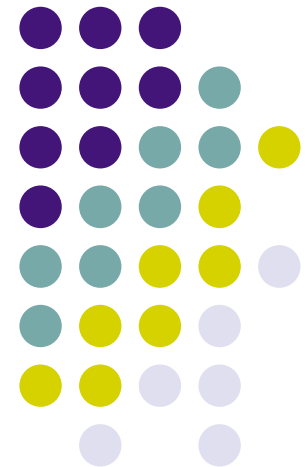
- The evaluation of `value` is triggered when `result` is needed
- If the evaluation of `value` returns an exception, then we wrap the exception in a **failed value** using the Ozma primitive `makeFailedValue`
- This method is actually native in the Ozma implementation for efficiency

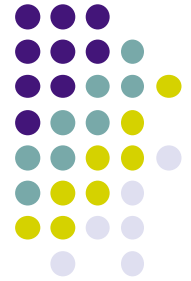


More on Scala concurrency

- Scala already has a rich set of concurrency abstractions
 - In Ozma these can be used together with dataflow
- Scala's concurrency abstractions are designed to be efficient using the underlying Java mechanisms
 - The primitive mechanism: monitors with wait, notify, notifyAll
 - They are not good enough to implement Ozma, unfortunately
- Some of them are actually not bad at all:
 - Futures, syncvars, fork-join (deterministic!)
 - Asynchronous/synchronous channels (deterministic!)
 - Actors with mailboxes (comparable to Erlang)
- Even better concurrency is available in the Akka library
 - Dataflow variables and transactional actors

The Past is Prologue





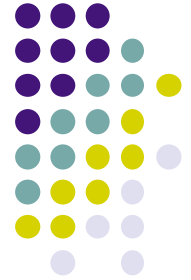
Whither Ozma?

- Ozma makes concurrent programming simpler
 - The heart of a concurrent program is deterministic. Nondeterminism is added just where it's needed.
 - Correctness is easy: the deterministic part is purely functional and the nondeterministic part uses message passing
- The Ozma implementation uses the Oz virtual machine (Mozart)
 - It's a complete implementation of Scala on a new VM that's not the JVM or .NET, so you can see it as a **new implementation of Scala**
 - It's not interoperable with Java, though. The Mozart VM was used because of its support for fine-grain threads, dataflow, and failed values.
- We are thinking about the future of Ozma. Would you be interested in a supported version? Should we join the Scala community and work on Scala's concurrency model? Or should we join the Java community and work on the JVM (like Flow Java did)?

Generalizing dataflow for distribution and fault tolerance



- Language support for distributed programming in Oz
 - **Network transparency**: a program executed over several nodes gives the same result as if it were executed on a single node, provided network delays are ignored and no failure occurs
 - Exact same source code is run independent of distribution structure
 - **Network awareness**: a program can predict and control its physical distribution and network behavior
 - Fully implemented in Oz (Mozart 1.4.0)
- Modular fault tolerance in Oz using fault streams
 - ✗ **Exceptions and RMI**: synchronous, not modular, requires changing code at each possible distribution point
 - ✓ **Fault streams on language entities**: asynchronous, modular, just add new code with no changes to existing code



Thanks for your attention!

- Ozma was developed as part of our research in programming languages and distributed computing
- We are currently hiring new Ph.D. candidates!
- Doctoral fellowships available in **Distributed Computing**
 - Erasmus Mundus program: see www.emjd-dc.eu
 - Salary + benefits (medical insurance)
 - Application deadline Jan. 23, 2012