A New Concurrency Model for Scala Based on a Declarative Dataflow Core

Sébastien Doeraene
École Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland
sebastien.doeraene@epfl.ch

Peter Van Roy
Université catholique de Louvain
Place de l'université 1
1348 Louvain-la-Neuve, Belgium
peter.vanroy@uclouvain.be

ABSTRACT

Declarative dataflow values are single assignment variables such that all operations needing their values wait automatically until the values are available. Adding threads and declarative dataflow values to a functional language gives declarative concurrency, a model in which concurrency is deterministic and explicit synchronization is not needed. In our experience, this greatly simplifies the writing of concurrent programs (as explained in several chapters of CTM [20]). We complete this model with lazy execution and message-passing concurrency. Both extensions are tightly integrated with the declarative dataflow core. Lazy execution is provided by extending declarative dataflow with a by-need synchronization operation. Message passing is provided by adding streams equipped with a `send` operation, where a stream is a list with an unbound single-assignment variable. This paper presents the Ozma language, a conservative extension of Scala that supports all these concepts. We have made a complete implementation of Ozma by combining the implementations of Scala and Oz. Evaluation shows that this implementation supports the full semantics of Scala with concurrent programs based on the new concurrency model. In particular, within the functional subset of Scala the new concurrency model fully supports deterministic concurrency.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages, data-flow languages*

General Terms

Languages

Keywords

Ozma, Scala, Oz, deterministic concurrency, dataflow, concurrent programming, lazy execution, message-passing concurrency, nondeterminism

1. INTRODUCTION

Programming in concurrent, parallel, or distributed settings always poses the same problem: how to synchronize access to shared data by several concurrent computations? One powerful solution is declarative dataflow, in which variables are single assignment and all operations that need values wait until the variables are bound. This paper presents Ozma, a conservative extension to Scala that adds declarative concurrency with single assignment dataflow values. Ozma turns every `val` declaration of Scala into a dataflow value, and every `var` into a mutable container that holds a dataflow value. This is so close to the existing Scala language that all existing Scala code can be used as part of a concurrent program, without any code change. This allows Ozma programs to take advantage of the full power of declarative concurrency with existing Scala libraries.

When used within the functional subset of Scala, a program executing within this model without raising any exceptions is deterministic. This is both a simplification and a limitation. It simplifies concurrent programming because race conditions are not possible. This follows from the use of single-assignment variables: in a concurrent program, what changes is not *what* value is assigned, but rather *when* the value will be assigned. However, this simplification comes with a strong limitation: programs that require nondeterminism cannot be expressed. For example, a simple client-server application with two clients and one server cannot be written in the declarative dataflow model. To overcome this limitation, we add a nondeterministic operation, namely communication channels with an asynchronous send operation. We use a simple form of channel called a *port*, which is simply a stream equipped with a `send` operation, where a stream is a list value whose tail is an unbound dataflow value. Ports allow to introduce nondeterminism exactly where needed, since any thread that knows the port's reference can send to the port, which appends the element to the port's stream.

To complete the concurrency model, we extend the declarative dataflow model with lazy execution. Lazy execution is supported by a second synchronization operation, `waitNeeded`, that waits until a dataflow value is needed by an operation. The combination of the two synchronization operations, namely waiting for a value and waiting until a value is needed, gives a declarative model that combines both concurrency and lazy execution.

Programming methodology based on the new model.

It may seem unusual to base a concurrency model on a deterministic concurrent core, especially considering that concurrent programs that interact with the real world need to model nondeterminism. But this is no different than basing a programming model on a deterministic *functional* core, despite most real-world applications needing some non-pure computations. Yet languages with a functional core are widely accepted, because most of the program can still be written in a functional style. To motivate the structure of the Ozma concurrency model, we outline briefly how we can apply the same methodology and layering of computing models to construct general concurrent programs.

The concurrent program starts as a sequential functional program that calculates with lists and other functional data structures. Note that it can be proved correct at this stage using standard proof techniques of functional programs. We modify the initial program in three steps to obtain the final program. First, we introduce concurrency exactly where needed by adding `thread` expressions. Second, we introduce nondeterminism exactly where needed by replacing input lists with ports (recall that ports are simply streams, i.e., lists, with a `send` operation). Finally, we introduce laziness exactly where needed by adding `waitNeeded` instructions in front of calculations that are optional, i.e., whose result is only needed in certain circumstances. The final program combines concurrency, nondeterminism, and laziness, exactly where needed. The advantage of this methodology is that the use of nondeterminism (and consequently mutable state) is minimized to only those places where it is needed. For example, in the client/server application mentioned before, nondeterminism is needed in one place only, namely the point at which the server receives a request from one of several clients. The rest of the application can be written in a completely functional style, if desired, without limiting the client/server functionality.

Note that each of these steps *does not invalidate the correctness proof* of the initial program, but just changes the conditions under which this proof is applied (see Section 3 for explanation why this is so). In case of true multiagent collaboration (i.e., concurrent entities working together to maintain a global invariant), an additional proof is needed for the collaboration. In simple cases, such as a client/server application in which the clients interact independently with the same server, this additional proof is not needed.

Natural extension of Scala.

This paper presents the design, implementation, and evaluation of Ozma, an extension of Scala that supports a powerful concurrent programming model based on a declarative dataflow core. Ozma provides a new set of concurrent programming techniques that naturally extend the existing concurrency model of Scala. Scala has an Erlang-style actor process model, where actors are objects running concurrently to which messages can be sent [11]. Each actor has a mailbox of incoming messages represented as a queue. This concurrency model is well-integrated with the Scala object model, however it is conceptually quite different from a functional language. This is where Ozma completes the Scala model: the declarative dataflow model of Ozma is a natural extension of the functional style in Scala, just as the actor model is a natural extension of the object-oriented style in Scala. We say that the new concurrency model “lifts” the duality of Scala (functions versus objects) to concurrent programming (declarative concurrency versus message-passing concurrency).

Implementation.

Ozma is compiled with a modified version of the Scala compiler and runs on the Mozart Programming System, which implements Oz [14, 13]. Full source code for Ozma is available with an open-source license on GitHub [3]. This implementation was not designed for performance, but for completeness: it is an implementation of the complete Scala language. Ozma can therefore be considered as a new implementation of Scala, in addition to the two existing implementations running on the JVM and .NET. The Ozma implementation demonstrates that declarative dataflow concurrency is possible and sound within Scala. Future developments will enhance the performance of Ozma by improving the data representation; we expect that performance similar to that of Oz on Mozart can be achieved.

Structure of the paper.

Section 2 compares Ozma with similar language design efforts. Section 3 presents the concurrency concepts of Ozma. Section 4 follows this by giving a precise semantics of Ozma as an extension of Scala semantics. Section 5 explains how Ozma was implemented as a combination of the Scala and Oz implementations. Section 6 evaluates Ozma according to several criteria, namely correctness, coverage, and performance. Finally, Section 7 concludes by briefly summarizing the work and its future development path.

2. RELATED WORK

Both deterministic concurrency and message-passing concurrency have a long history. Two of the earliest references of each are Kahn process networks from 1974 [10] and Hewitt’s Actor Model from 1973 [7]. See the textbook [20] for more history and related work up to 2003. Since the 1970’s, several attempts have been made to add forms of deterministic concurrency to existing languages.

The concept of declarative dataflow synchronization has existed since the early 1980s. It was introduced in logic programming through the Parlog and Mu-Prolog languages, and was then extended to a full-blown concurrent execution model in Concurrent Prolog, which was used in the Japanese Fifth Generation Computer project in the 1980s [17]. It was introduced independently in functional programming through the concept of I-structures, which was influential in hardware dataflow architectures [1].

Another attempt to add a form of deterministic concurrency to programming is MultiLisp, a language intended for writing parallel programs [6]. MultiLisp extends Lisp with a concept called a *future*. The function call `(future E)` where `E` is an expression does two things: it initiates a concurrent evaluation of `E` and it immediately returns a placeholder for the result of `E`. When another evaluation needs the value of `E`, then it waits until the value is available.

The concept of future is becoming prevalent in Scala itself. Scala 2.10 introduces futures and promises at the library level: code that deals with asynchronous computations must do so explicitly. Habanero-Scala presents a form of delimited asynchronous computations for Scala [8]. Ozma goes a step beyond these localized extensions by making futures ubiquitous and transparent through dataflow values.

The language that has inspired Ozma the most is Oz, as defined in [20]. Oz is based on the concurrent constraint programming model [16] and was originally designed by Gert Smolka and his students [18]. The port concept originated in AKL [9] and the by-need synchronization (`waitNeeded`) originated as an extension for declarative laziness [19]. All three of the new concepts in Ozma, namely dataflow synchronization, by-need synchronization, and ports, have corresponding concepts in Oz. The `@tailcall` transformation was also inspired by Oz, but we have generalized it in Ozma to support user-defined methods, which was of the utmost importance since all structured values are user-defined in Scala. Ozma brings all these features into Scala, and shows that they can be made to fit perfectly well in a statically typed, object-oriented language that also has its share of mutable, imperative-style features.

Another language similar to Ozma is Flow Java [5], which conservatively extends Java with single assignment variables and dataflow synchronization. There are several differences between Ozma and Flow Java. Flow Java adds only declarative dataflow variables to Java and does not have any extensions for message passing nor for lazy execution. Flow Java is an extension of Java, which does not have an obvious declarative subset, whereas Ozma extends Scala, which does have a straightforward declarative (functional) subset. This means that it is much easier to write declarative concurrent programs in Ozma than in Flow Java. Finally, the run-time systems are quite different. Flow Java runs on top of a modified JVM with support for single assignment variables, whereas Ozma uses the existing Mozart Programming System. This means that Ozma can take advantage of the lightweight threads of Mozart, whereas Flow Java must use the Java thread implementation.

Finally, we mention the language Alice ML [15], which is a language that combines many ideas from Oz and Standard ML. Alice ML has single assignment variables (without aliasing), futures, and ports. Like Flow Java, it runs on a dedicated virtual machine. Like Oz, Alice ML was not designed specifically for deterministic concurrency, and like Oz it supports this model because of its well-factored design.

3. CONCURRENT PROGRAMMING IN OZMA

This section presents the new concurrency concepts that Ozma adds to Scala. We give examples for each concept and we explain the strong property that each concept satisfies. Precise semantics of these concepts is given in Section 4. We assume basic familiarity with Scala for the rest of the paper. Before introducing the concurrency concepts, we explain briefly what we mean by declarative concurrency.

3.1 Declarative concurrency

Before showing the concurrency techniques for Ozma, we first explain what it means for the concurrent dataflow model to be declarative and under what conditions it is declarative. The concurrent dataflow model consists of functional programs extended with dataflow values and synchronization on dataflow values. To obtain declarative execution in Ozma, we restrict programs to use only functions with `val` declarations and we consider only executions that raise no exceptions. Mutable assignment, such as `var` declarations, is forbidden in the functional subset. It is clear that concurrency together with mutable assignment or exceptions can introduce nondeterminism, and hence is not declarative. Intuitively, a concurrent dataflow program is declarative if there is no observable determinism. To make this precise, we give the following definition:

Definition 1 (Declarative concurrency). *A concurrent dataflow program is* declarative *if for all possible inputs the following holds. For all possible executions with a given input, one of two results is possible. (1) They all do not terminate, or (2) they all eventually reach partial termination and give results that are logically equivalent.*

To make the above definition self-contained, we define the four new concepts it uses:

Definition 2 (Dataflow program). *A* dataflow program *is a function with one input argument and one output argument, where the input and output arguments are partial values. All calculations are done with dataflow values: all results of function evaluations are bound to dataflow values and all function arguments are dataflow values. Any function that needs its argument to be a value (constant) will wait (not reduce) until the argument is bound to a value (dataflow synchronization).*

The restriction to a single input and output argument does not lose generality, since it is always possible to group inputs and outputs together inside compound values (e.g., tuples or other instances of immutable classes).

Definition 3 (Partial value). *A* partial value *is either an unbound dataflow value or a dataflow value bound to a primitive value, or an instance of a class such that each field of this instance is also a partial value.*

Partial values are an essential part of the execution of a dataflow program. For example, an important partial value is the stream, which represents a communication channel. A stream is a list with an unbound dataflow value as its tail. Extending the tail by binding it to a cons cell corresponds to sending an element on the channel.

Definition 4 (Partial termination). *We say that a dataflow program exhibits* partial termination *with a given input, if for that input the execution of the program eventually stops (no more execution steps are possible). The output after partial termination is a partial value.*

Partial termination generalizes complete termination: a program that is partially terminated may continue execution if the input argument is further bound, i.e., if any unbound dataflow values in the input are further bound. For example, a program whose input and output are streams may have an unbounded number of partial terminations, namely each time an element is added to the input stream.

Definition 5 (Logical equivalence). *Given two executions of the same dataflow program with the same input and the same initial unbound dataflow value as output. When the two executions reach partial termination, we say that the two outputs are logically equivalent if they are identical up to variable renaming. That is, it is possible to rename the variables in one output to be identical to the other output.*

For example, if the two outputs are `1 :: x` and `1 :: y` where `x` and `y` are unbound dataflow values, then the two executions are logically equivalent.

Given these definitions, we say that an Ozma program that is defined using the language subset containing only fair threads, functions, values (including instances of immutable classes), and `val` declarations, is declarative concurrent if during its execution it raises no exceptions. Note that the converse is not true. There exist Ozma programs using, e.g., `var` declarations, that are declarative concurrent nevertheless. We now outline a proof of this property. A formal proof is yet to be done, and is the subject of future work.

Consider any sequential execution of a pure functional program that does not raise exceptions. Such an execution is equivalent to an execution of the λ-calculus and is therefore confluent: all possible evaluation orders give the same result. In the relevant subset of Ozma, this execution is expressed using dataflow values and is therefore a dataflow program. Each dataflow value has three well-defined points in its lifetime: declaration, binding, and use. We assume without loss of generality that a dataflow value has a single declaration, a single binding, and possibly multiple uses. In any execution without exceptions, multiple bindings will always be to the same value. Multiple bindings to different values will raise an exception on all values different from the first binding, and hence will not occur. Consider now the dataflow value `x` in a sequential execution:

```
...
val x: Int // declaration of x
...
x = 23     // binding of x
...
y = x+1    // first use of x
z = x+2    // second use of x
...
```

For simplicity, this example binds `x` to an integer, but the same reasoning applies to all values. Now consider a concurrent execution with the same operations as the above sequential execution. The concurrent execution is obtained by executing an arbitrary number of function applications in their own threads. This will delay all the operations in the evaluation of these functions and the binding of the result. The declaration, binding, and uses of `x` may therefore be delayed (occur in a later execution step) with respect to the sequential execution, however, the order of these operations respects two rules. First, the declaration will always occur before the binding and uses. This follows because of causality: a function is only invoked with arguments that exist. This is true whether or not the function is invoked in its own thread. Second, all uses will always occur after the binding. This follows because any attempt to use `x` before binding will cause the use to be delayed until the binding occurs (dataflow synchronization). Because the dataflow value is single assignment, and because it is bound to the same value as in the sequential program, we conclude that all uses in the concurrent program will see the same binding as in the sequential program. The result of the concurrent program is therefore the same as the sequential program.

3.2 Adding concurrency to any functional computation

In Ozma, any functional computation can be made concurrent, while preserving the final result, by introducing calls to `thread`. Moreover, such a concurrent program is guaranteed not to have any race condition. This is the most powerful property of declarative concurrency in Ozma: concurrency is *deterministic.*

All values are implicitly *dataflow values.* A dataflow value is similar to a regular value, except that it can be *unbound.* Unbound values are not initialized yet, but can be aliased to other unbound values. When a dataflow value gets bound, all its aliases get bound to the same value, automatically. Moreover, statements requiring a value to be bound *wait* for it to be bound instead of crashing, providing automatic builtin synchronization.

Initially unbound values can be declared explicitly. However, they are generally introduced implicitly through the use of the primitive `thread`,

```
def thread(body: => A): A
```

which can be used like in

```
val x = thread(someComputation())
```

This primitive spawns a thread and immediately returns an unbound value. This value will get bound to the result of the threaded computation when it is finished. This behavior is similar to *futures*, e.g., in MultiLisp, but it is more general since the concepts of dataflow value and thread are separated.

Adding concurrency in this manner can be seen as rearranging the calculations of a program while keeping the causality property of each value binding, i.e., the value is calculated before it is used. It is clear that this does not change the result of a calculation, since by induction the variable is still bound to the same value, only at a different point in the execution.

Example: concurrent map.
Consider the `map` function on lists,

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

which is a deterministic sequential computation. It can be turned into a deterministic *concurrent* computation simply by wrapping the call to `f` with a thread:

```
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)
```

The resulting `concMap` is *deterministic* and its result is always the same as the sequential version. However, all applications of `f` execute concurrently. Because dataflow variables are really ubiquitous, `concMap` can also reuse `map`, the latter suddenly manipulating dataflow variables although it was not designed with this in mind:

```
def concMap[A, B](list: List[A], f: A => B): List[B] =
  map(list, x => thread(f(x)))
```

3.3 Converting any list function into a declarative agent

Any function computing on lists that is tail-recursive modulo cons can be turned into a long-lived declarative concurrent agent, without memory leak, simply by executing it in a thread. Here we define an *agent* as an active entity that processes an input stream and/or produces an output stream. We define a *stream* to be a list whose tail is unbound. Streams can grow indefinitely. They can be used as communication channels between agents. One thread is processing the stream using standard list operations, while another thread is building the list. Communication is achieved between the two threads by sharing the dataflow value that references the stream.

To support this, Ozma implements a tail call optimization for functions that are tail-recursive modulo cons, such as map. It follows that any such function executes in constant stack space and can run indefinitely without memory leak. Section 4.7 explains the compiler transformation used to achieve this.

Example: filter and map as agents.
The following code snippet is a simple program that displays squares of even numbers on the standard output.

```
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
}

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

Wrapping the calls to `gen`, `filter` and `map` within threads is sufficient to turn them into declarative agents. Note that `foreach` is also an agent, living in the main thread. Declarative agents are incremental: as new elements are added to an input stream, new computed elements will appear on the output stream.

Although the call to `gen` never terminates, the `@tailcall` transformation described in Section 4.7 ensures that it produces partial results, which can be consumed concurrently by the `filter` agent. The same applies to the other agents.

3.4 Adding laziness to any computation

Any computation, whether functional or not, can be made *lazy* by introducing `waitNeeded` statements. This means that the computation will be deferred until the value is *needed*, i.e., until an instruction waits for the value. If this never happens, then the computation is never executed.

The `waitNeeded(x)` statement blocks the current thread until `x` becomes needed. Any number of `waitNeeded` calls can be added to a program, and the program will give the same result as before, provided no deadlock is introduced. Deadlocks appear when there is a `waitNeeded(x)` call that prevents the variable `x` to become needed (a circular need dependency). The combination of declarative concurrency and `waitNeeded(x)` retains the property of being deterministic. Calls to `waitNeeded` can be added in just the right places to add exactly the degree of laziness desired.

There are two common patterns for `waitNeeded` that are given their own names. `byNeedFuture` can be used as a wrapper, like `thread`, to make its body be lazily computed. `.lazified` can be used as prefix on list methods to obtain a lazy version. These two patterns never introduce deadlocks on their own, since by construction they cannot introduce circular need dependencies.

Example: lazy filter and map as agents.
The previous example can be made lazy by introducing `byNeedFuture` wrappers and `.lazified` prefixes. Explicit thread creation is removed, since it is implied by these two constructs.

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

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

Now `foreach` imposes the control flow. Laziness prevents the agents from getting ahead of the consumer. This provides better memory management, since it guarantees that the whole program executes in constant memory space.

3.5 Managing nondeterminism with ports

A port is a referenced stream, where the reference is a constant value. Sending a value to a port will append it to the stream. Ports are used to add nondeterministic behavior to a concurrent program exactly where it is needed. The readers and writers of a port can themselves be deterministic computations. The only source of nondeterminism is the order in which values sent to a port appear on the port's stream.

This model permits to combine deterministic concurrency with more traditional nondeterministic concurrency. This is a powerful approach to write concurrent programs that deserves more recognition (see chapter 4 and subsequent chapters of [20]). In our experience writing realistic programs, few nondeterministic constructs are necessary, which makes it much easier to show the correctness of the concurrent program.

A port is created using the primitive `Port.newPort`, which returns a port and a stream. Sending an element to the port through its `send` method appends the element to the end of the stream.

Example: partial barrier synchronization.
A barrier is an abstraction that starts n tasks concurrently and then synchronizes on their completion. In a partial barrier, we wait for only $m \leq n$ tasks to be completed. The following code snippet implements partial barrier synchronization with a port, whose stream contains the results of completed tasks. The function returns the results of the m first tasks that are completed. The call to `take(stream, m)` implicitly waits for m elements to appear on the stream.

```
def partialBarrier[A](m: Int,
    tasks: List[() => A]): List[A] = {
  val (stream, port) = Port.newPort[A]
  for (task <- tasks)
    thread { port.send(task()) }
  take(stream, m) // take the first m elements
}

def take[A](s: List[A], m: Int): List[A] =
  if (m == 0) Nil
  else s.head :: take(s.tail, m-1)

val M = 1 // or 2 or 3
val results = partialBarrier(M, List(
    () => { sleep(1000); println("a"); "a" },
    () => { sleep(3000); println("b"); "b" },
    () => { sleep(2000); println("c"); "c" }
))
println(results)
```

This displays "List(a)" right after displaying "a". Setting M to 2 changes this behavior so that "a" and "c" are displayed before "List(a, c)".

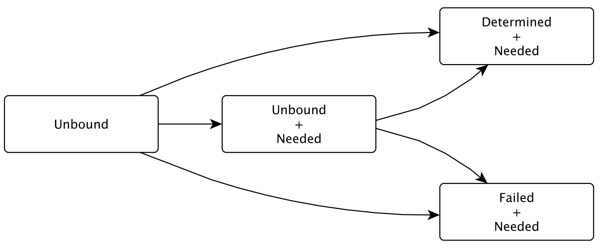

Figure 1: Possible status transitions for dataflow values

4. SEMANTICS

Ozma is a conservative extension of Scala: all legal Scala programs are also legal Ozma programs. Therefore it suffices to give an extension of the Scala language specification that makes precise the semantics of Ozma's extensions [12]. Since Scala has only a natural-language semantics and no formal semantics, it is impossible to give a completely formal semantics of Ozma. However, since Ozma is related to Oz, which does have a formal semantics, we refer interested readers to [20, chapter 13].

4.1 Value status

The most important semantic change introduced in Ozma is the dataflow value. In Ozma, every `val` is a dataflow value, and every `var` is a mutable container for a dataflow value. Dataflow values are defined by their *status*. At any point during a program's execution, a dataflow value must have exactly one of the following statuses: unbound (which is the initial state of all values), unbound and needed, determined and needed, or failed and needed. A value is said to be bound when it is not unbound, i.e., when it is determined or failed. The status of a dataflow value can change over time, but is monotonic. The possible status transitions of a value are shown in Figure 1.

When a value is determined, it holds the *actual value* it has been bound to. An actual value is either a primitive value (integer, float, boolean, character, or unit) or a reference value (`null` or a reference to an object).

When a value is failed, it holds a throwable value (whose type conforms to `Throwable`). The dynamic type of such a wrapping is `Nothing` (the *bottom* type of Scala), so that it conforms to any type and can be stored in any value.

In addition, each value is associated with a set of threads waiting for the value to be *bound* (the wait-for-bound set) and a set of threads waiting for it to be *needed* (the wait-for-needed set).

4.2 Primitive operations on value status

Initially, a value is unbound, and the two wait-for sets are empty. The following primitive operations are defined on a value x. All operations on a value are atomic, unless and until they suspend the current thread.

`waitBound(`x`)` waits for x to be bound.

- If x is unbound (needed or not): add the current thread to the wait-for-bound set of x, set the status of x to unbound and needed, and suspend the thread.
- If x is determined: do nothing and return immediately.
- If x is failed: throw the exception wrapped inside x.

`waitQuiet(`x`)` is a variant of `waitBound(`x`)` that does not modify the status of x, i.e. it does not mark x as needed. It also returns normally if x is a failed value.

`waitNeeded(`x`)` waits for x to be needed.

- If x is not needed (and a fortiori unbound): add the current thread to the wait-for-needed set of x, and suspend the thread.
- Otherwise: do nothing and return immediately.

$x = y$ unifies x and y.

- If either x or y is unbound (or both): make x and y *aliases of each other*, with the more specific of the statuses of x and y (the farthest to the right in the schema), and the union of the sets waiting for them. From that point on, x and y share their status, and any operation on x applies to y and conversely. If the resulting value is needed (resp. bound), resume all threads in the wait-for-needed (resp. wait-for-bound) set and empty it.
- If both x and y are failed: throw either the exception wrapped in x or in y (the choice is nondeterministic).
- If one of x or y is failed and the other is determined: throw the exception wrapped in the failed value.
- If both x and y are determined: check the two values for reference equality (the *eq* method). If x eq y, do nothing. If x ne y, throw a failure exception.

The two following primitive functions build bound values.

`makeDeterminedValue(`v`)` creates a new determined value that holds the actual value v, and returns that value. v can be a literal constant, a `new` instance, or the result of a primitive arithmetic or logic operation. This primitive is not available at the user level.

`makeFailedValue(`x`)` creates a new failed value that holds the throwable x, and returns that value. The static type of x must conform to `Throwable`.

4.3 Builtin synchronization

A number of language constructs implicitly wait on unbound values. That is to say, they behave as if they first did a `waitBound(x)`. The following operations behave this way:

- Calling a method with receiver x waits for x.
- Boolean `||` and `&&` block on their left operand, and on their right operand if and only if the actual value of the first one could not determine the result of the operation.
- `eq` and `ne` normally wait for their operands, but can continue without blocking if the operands are aliases of each other.
- Other comparisons, boolean operations, and arithmetic operations wait for both their operands.
- `if` statements, `while` and `do..while` loops wait for their condition each time it is evaluated.
- `match` statements wait for the expression to match.

Example: integer addition.
The method `Int.+(x: Int): Int` is typically considered as a primitive. In Ozma, it is a little more than a primitive, since it provides automatic synchronization. It can be defined as follows:

```
def Int_+_Int(x: Int, y: Int): Int = {
  waitBound(x)
  waitBound(y)
  actual val v = primitive_Int_+_Int(x, y)
  makeDeterminedValue(v)
}
```

The other language constructs listed above have similar definitions.

4.4 Status of boxed values

In Scala, values of primitive types are sometimes *boxed* and then *unboxed*. This happens when they are bound to a value of type `Any` or of a generic type. This is also the case in Ozma. Boxing and unboxing, being supposedly transparent for the user, raise an issue regarding unbound values, because they should respect transitivity of aliasing. It is tempting to define aliasing so that it follows boxes naturally. This would solve the problem at the semantics level, but would be impractical to implement efficiently.

We define the behavior of boxing and unboxing operations as follows. For a boxing or unboxing of the `Unit` value, the operation always immediately returns a determined boxed/unboxed `Unit`, whatever the status of the argument is. For all other primitive types, the results of the operations `y = box(x)` and `y = unbox(x)` follow these rules.

1. If x is determined, then y is immediately determined and holds the corresponding boxed or unboxed actual value (as defined in the Scala specification).
2. If x is failed, then y is immediately unified to x, i.e., it becomes the same failed value.
3. If x is unbound, then y is unbound and, at all times:
   (a) if x is needed, then eventually y is needed,
   (b) if y is needed, then eventually x is needed,
   (c) if x is bound, then eventually rule 1 or 2 applies.

4.5 Lazy execution

Lazy execution is built on top of the primitive `waitNeeded`. We define the function `byNeedFuture` as a convenient wrapper to declare values that should be computed only when they become *needed*.

```
def byNeedFuture[A](body: => A): A = {
  val result: A
  thread {
    waitNeeded(result)
    result = try body
             catch { case th => makeFailedValue(th) }
  }
  result
}
```

in which the `try...catch` ensures proper exception propagation between threads, through the use of failed values. It is often used to wrap the body of a function:

```
def someComputation(x: Int) = byNeedFuture {
  x + 1
}
```

Lazy execution in Ozma is more powerful than lazy values in Scala. In Scala, a lazy value is evaluated as soon as it is *accessed* for the first time. In Ozma, the result of a lazy computation is an unbound dataflow value, which can be aliased to other values and passed around through the program. It gets evaluated only when the value is actually *needed*.

4.6 Ports

Ports consist of two primitives: `newPort` and `send`. `newPort` creates a new port handle, and returns a pair of (a) the stream for use by the consumer and (b) an instance of `Port` for use by the producers. The stream is initially unbound, and the `Port` instance has a reference to it. This reference is internally mutable.

```
class Port[-A] private (...) {
  def send(element: A): Unit
}

object Port {
  def newPort[A]: (List[A], Port[A])
}
```

Calling the port's `send` method, `port.send(element)`, executes the three following operations as one atomic operation:

- Create a new unbound value `tail` of type `List[A]`,
- Bind the stream referenced by the port to a new list pair `element :: tail`,
- Replace the reference of the port so that it points to the new tail, `tail`.

That is, `port.send(element)` appends `element` to `port`'s stream.

4.7 `@tailcall` transformation

This final section defines a transformation that the compiler is *required* to apply. This transformation is essential to guarantee the absence of memory leaks and even progress with various programming techniques of the model. For example, a special case of this transformation allows tail-recursive functions modulo cons to be tail call optimized, which was explained to be essential in Section 3.3. The `@tailcall` transformation is therefore an integral part of Ozma semantics.

Let there be a method m with parameters p_1 to p_n, and one or more parameters p_i are annotated with the `@tailcall` annotation, e.g.:

```
def m(p1: T1, ..., @tailcall pi: Ti, ...,
    @tailcall pj: Tj, ..., pn: Tn): T
```

Let A be the set of indices corresponding to `@tailcall`-annotated parameters. Here $A = \{i, j\}$.

Somewhere in the program (possibly in this particular method), there is a call to m in tail position, with actual parameters a_1 to a_n, e.g.

```
def someMethod() {
  doSomething()
  if (cond) m(a1, ..., ai, ..., aj, ..., an)
}
```

Let B be the set of indices corresponding to actual parameters that are themselves a call to a method (including accessor methods). An expression is a method call if and only if it is not one of (a) a constant literal, (b) a local value or variable, (c) a class literal `classOf[C]`, or (d) an arithmetic or logic operation. For example, in this code:

```
val someLocal = 5
m(5, someField, someLocal.toString, someLocal,
    classOf[String], someField+1)
```

$B = \{2, 3\}$ because `someField` is a call to the accessor method of `someField` and `someLocal.toString` is a call to the `toString` method. The four other parameters are examples of the four categories of expressions that are not method calls.

Now let $C = A \cap B$ be the set of indices that correspond to both a `@tailcall`-annotated formal parameter and a method call actual parameter. If $C = \varnothing$, then nothing special happens. If $C \neq \varnothing$, then let i be the *right-most* element of C, i.e., the maximum element:[1]

$$i \in C \wedge \forall j \in C : i \geq j \ .$$

By definition of B, `ai` is of the form `meth(params...)`. Then the call to m is replaced by a block equivalent to

```
{
  val arg: Ti
  result = m(a1, ..., arg, ..., an)
  meth(params..., arg)
}
```

where `result` is an output parameter that takes the role of the return value. This transformation makes the call to `meth` be *tail call*, thereby allowing for further tail call optimization with usual techniques. Note that this is valid only if m can execute without blocking on `arg`, which is true for list creations.

In the last method call, `arg` is passed `meth` as an *output* parameter. This is not valid user-level Ozma code, it is internal only. Likewise, all methods are internally rewritten as taking an output parameter rather than returning a value. For example, the method `meth`, initially defined as

```
def meth(params...): R = body
```

would be rewritten in the following internal form:

```
def meth(params..., out result: R): Unit = result = body
```

`@tailcall` *and case classes.*

The `@tailcall` annotation can be used to mark some or all of the parameters of a method as *unbound safe*. This means that the method can execute and return without blocking when that parameter is unbound. Because the `@tailcall` transformation is most useful with case class constructors, they receive particular attention from the compiler, so that the user need not state the annotation explicitly. When defining a case class *without any constructor code*, all the parameters of the constructor are automatically `@tailcall`-annotated by the compiler. In the following code snippet, which defines a stateless binary tree of integers,

```
abstract sealed class Tree
case object Leaf extends Tree
case class Node(value: Int, left: Tree,
    right: Tree) extends Tree
```

the parameters `value`, `left` and `right` of the constructor of `Node` automatically receive the `@tailcall` annotation. This allows the following implementation of insertion with conservation of ordering to be `@tailcall`-transformed, so that it becomes fully tail-recursive:

[1]This strategy is arbitrary: other priority conventions may be used instead. We chose right-to-left priority because *usually*, right-most parameters are more subject to recursion than others. A trivial example is the `::` class where recursion always happens on the tail of the list, which is the right-most parameter.

```
def insert(tree: Tree,
    value: Int): Tree = tree match {
  case Leaf => Node(value, Leaf, Leaf)
  case Node(v, left, right) =>
    if (value <= v)
      Node(v, insert(left, value), right)
    else
      Node(v, left, insert(right, value))
}
```

The transformation gives the following result:

```
def insert(tree: Tree, value: Int,
    out result: Tree) = tree match {
  case Leaf => result = Node(value, Leaf, Leaf)
  case Node(v, left, right) =>
    if (value <= v) {
      val arg: Tree
      result = Node(v, arg, right)
      insert(left, value, arg)
    } else {
      val arg: Tree
      result = Node(v, left, arg)
      insert(right, value, arg)
    }
}
```

which is clearly tail-recursive. In the standard library, the following case classes have `@tailcall` annotations: `::` (subclass of `List`), `Some` (subclass of `Option`), `Left` and `Right` (subclasses of `Either`), and all tuple classes.

Example.

In the previous code snippet, the last two calls to the constructor of `Node` fulfill the requirements for `@tailcall` transformation. Indeed, in both cases, $A = \{1, 2, 3\}$ since all parameters are `@tailcall`-annotated. m is the `insert` method, and `meth` is successively the two recursive calls to `insert`. In the first case, $B = \{2\}$, because `v` and `right` are local values, hence $C = \{2\}$, and $i = 2$. In the second case $B = \{3\} = C$, and $i = 3$. Since $C \neq \varnothing$, the transformation applies. This gives the following transformation. A first rewriting to internal form makes the result explicit through an output parameter.

```
def insert(tree: Tree, value: Int,
    out result: Tree) = tree match {
  case Leaf => result = Node(value, Leaf, Leaf)
  case Node(v, left, right) =>
    if (value <= v)
      result = Node(v, insert(left, value), right)
    else
      result = Node(v, left, insert(right, value))
}
```

Then, the two last statements are rewritten using the tail call transformation.

5. IMPLEMENTATION

Unlike the standard Scala implementation, Ozma runs on the Mozart VM instead of the Java VM. We chose this design because the Mozart system, which implements Oz, has efficient support for lightweight threads and dataflow values. This design decision has several important consequences.

Scala is defined to be compilable to efficient code on the JVM. However, the Mozart VM behaves very differently. For example, the object models of the two VMs differ significantly. The implementation of Ozma reduces the gap between them by providing an encoding scheme of Java-like classes for use in Oz code. This encoding adds support for the meta-object protocol, method overloading, classpath-based class loading, and so forth.

The Ozma compiler.
The Ozma compiler is written in Scala as a pure extension via subclassing of `nsc`, the official Scala compiler. `nsc` is organized as a series of *phases* that successively transform the Abstract Syntax Tree (AST) towards a simple form that can be compiled easily into JVM bytecode. In the Ozma compiler we modify the set of phases to support dataflow values, and we replace the entire back-end so that it outputs an AST for Oz code instead of JVM class files. The Oz AST is then compiled into Mozart bytecode by the standard Oz compiler.

Compiling single assignment values.
The compilation of single assignment values has consequences in several phases. In Scala, declaring a local value without initializing it is illegal. An early phase of the Ozma compiler, `singleass`, patches such values so that they can get through the rest of the compiler without problem. Any single assignment value, of the form:

```
val value: Type
```

is patched by this phase to become:

```
@singleAssignment var value: Type =
  scala.ozma.newUnbound[Type]
```

It is difficult to get past error checking in subsequent phases of the compiler, which do not know about single assignment values. We mark the value as *mutable* (i.e. being a `var`) so that assignment to this value is considered valid. In order for the back-end to remember that it has to be compiled as a single assignment value, we give it a dedicated annotation, namely `@singleAssignment`. Finally, we give it a dummy right-hand-side to bypass some error checking.

Single assignment values being known as variables by the rest of the compiler is actually fine, except for one phase. The `lambdalift` phase must be patched, by using a subclass and overriding some methods, in order to behave correctly in the presence of single assignment values.

6. EVALUATION

Correctness of the compiler.
We tested the compiler for correctness and absence of memory leaks on a large set of examples covering Oz programming techniques that were translated to Ozma [3]. We have also compiled and run the entire Scala library, which is used by these examples.

The current implementation is possibly faulty for some instances of `match` constructs. The earlier phases of the compiler transform these into *jumps*, which cannot be represented using Oz ASTs. We have written a partial translation function for such jumps, which compiles correctly the Scala library, but we are not convinced that it works for all ASTs produced by `match` expressions. A future implementation targeting Scala 2.10 could fix this, because jumps emitted by its new pattern matcher are much more regular and easy to translate.

Coverage of concurrency techniques.
Ozma supports all of the most useful concurrency features of Oz. Shared-state concurrency exists in Scala, so it exists also in Ozma. There are two concepts that we lose in Ozma with respect to Oz, namely full unification and distributed

	Ozma	Oz lists	Oz classes
GenList	12,636	50	309
MapFilter	26,342	318	1,229
ThreadedMapFilter	29,080	274	1,044

Table 1: Performance measurements (ms)

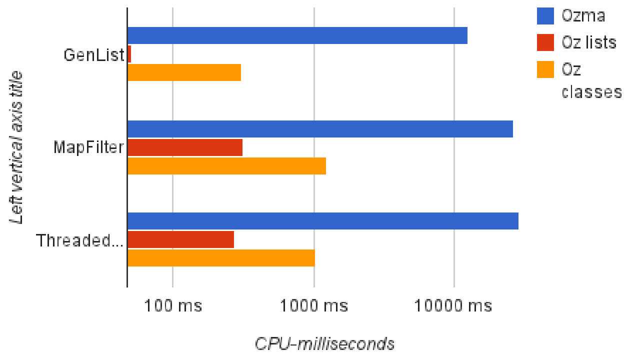

Figure 2: Performance measurements

programming. Full unification lets the aliasing operation dive inside functional data structures (named records in Oz) to any nesting depth. Ozma does not have records, but only objects. Experience shows that unification is only rarely needed for concurrent programming. Instead, unification is useful for constraint programming, which is supported by Oz. Distribution in Oz implements full network transparency and provides modular primitives for fault tolerance [2].

Performance.
The main goal of Ozma is support for concurrent programming, not performance. We focused our efforts towards providing a complete, working implementation of the entire Scala semantics and its Ozma extension.

Table 1 shows the execution time of three programs in three setups, and Figure 2 shows the same results using a logarithmic scale. GenList creates a list of 1,000,000 elements. MapFilter creates the same list, then applies a filter and a map on it. ThreadedMapFilter does the same, but in three different threads. The programs are written in Ozma, in Oz using standard lists, and in Oz using user-defined classes for lists. In each case, the experiment in run 50 times and averaged, on an Intel Core i7 CPU, 1,73 GHz, on Linux Debian Jessie 64 bits.

These results show that Ozma is 2 orders of magnitude slower than Oz using user-defined lists using classes. The latter being itself 1 order of magnitude slower than using native Oz lists. The Mozart VM aggressively optimizes built-in lists, which explains the second observation. Ozma is even slower because of the multiple indirections in the Scala library (which are inlined away in Scala/JVM, but not in Ozma).

In the two Oz programs, the threaded versions of MapFilter is slightly faster. This has nothing to do with parallelism, as the Mozart VM is single-core only. But in this case memory at the head of the intermediate lists can be reclaimed as the three agents move forward, which yields better performance of the garbage collection (which is a double-space-copy GC). The Ozma program exhibits slightly worse performance with the threaded version, however, although the reason is unclear.

The main performance issue is that Ozma uses Oz classes and objects for all data types, including lists, records and anonymous functions. Now, the Mozart VM aggressively optimizes native lists, records and anonymous functions, but does a very poor job at running heavily object-oriented code. Future development of Ozma should address this issue by compiling lists and anonymous functions down to their native counterpart.

7. CONCLUSIONS AND FUTURE WORK

We present a new language, Ozma, a conservative extension of Scala that offers a new concurrency model consisting of three parts: a declarative dataflow core with a message-passing extension and a lazy execution extension. Ozma is implemented as a combination of a Scala compiler front end and an Oz compiler back end and run-time system. Ozma extends Scala with deterministic concurrency, declarative agents, lazy execution, and nondeterministic agents using ports. These techniques support a methodology for building concurrent programs that starts with a correct functional program and successively adds concurrency, nondeterminism, and laziness exactly where they are needed. Future work includes improving performance, supporting fault-tolerant network transparent distributed programming, and porting to the JVM. Ozma has elicited interest from the Scala community; we mention in particular two invited talks at the QCon industrial software development conference [21], and at the Strange Loop conference [4]. We hope that the design of Ozma will have a positive influence on the future evolution of Java and Scala.

8. REFERENCES

[1] Arvind and Thomas R. E.: I-Structures: An efficient data type for functional languages. Technical Report 210, MIT, Laboratory for Computer Science, Cambridge, MA (1980)

[2] Collet, R.: The Limits of Network Transparency in a Distributed Programming Language. PhD thesis, Université catholique de Louvain (2007)

[3] Doeraene S.: Ozma: Extending Scala with Oz Concurrency. Master's thesis, Université catholique de Louvain. Full source code available at `https://github.com/sjrd/ozma` (2011)

[4] Doeraene, S.: Ozma: Extending Scala with Oz Concurrency. Invited talk, Strange Loop, St Louis, MI, URL: `thestrangeloop.com`, (Sep. 2012)

[5] Drejhammar, F., Schulte, C., Haridi, S., Brand, P.: Flow Java: Declarative concurrency for Java. In *Proceedings of the Nineteenth International Conference on Logic Programming*, Springer LNCS, vol. 2916, pp. 346–360. (2003)

[6] Halstead, R.H. Jr.: MultiLisp: A language for concurrent symbolic computation. In *ACM Transactions on Programming Languages and Systems*, 7(4), pp. 501-538 (Oct. 1985)

[7] Hewitt, C., Bishop P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 235-245 (Aug. 1973)

[8] Imam, S., Sarkar, V.: Habanero-Scala: Async-Finish Programming in Scala.

[9] Janson, S., Montelius, J., Haridi, S.: Ports for objects in concurrent logic programs. In *Research Directions in Concurrent Object-Oriented Programming*, pp. 211-231 (1993)

[10] Kahn, G.: The semantics of a simple language for parallel programming. In *IFIP Congress*, pp. 471-475 (1974)

[11] Odersky M.: Scala By Example. École Polytechnique Fédérale de Lausanne (2010)

[12] Odersky M.: The Scala language specification, version 2.9. École Polytechnique Fédérale de Lausanne (2011)

[13] Mozart Consortium: Mozart Programming System. URL: `www.mozart-oz.org` (2011)

[14] Programming Methods Laboratory: The Scala Programming Language. URL: `www.scala-lang.org` (2011)

[15] Rossberg, A.: Typed Open Programming: A Higher-Order, Typed Approach to Dynamic Modularity and Distribution. PhD thesis, Universität des Saarlandes (2007)

[16] Saraswat, V.A.: Concurrent Constraint Programming. MIT Press, Cambridge, MA (1993)

[17] Shapiro, E.: The family of concurrent logic programming languages. In *ACM Computing Surveys*, 21(3), pp. 413-510 (Sept. 1989)

[18] Smolka, G.: The Oz programming model. In *Computer Science Today*, Springer LNCS, vol. 1000, pp. 324-343 (1995)

[19] Spiessens, A., Collet, R., Van Roy, R.: Declarative Laziness in a Concurrent Constraint Language. In *2nd International Workshop on Multiparadigm Constraint Programming Languages*, part of *9th International Conference on Principles and Practice of Constraint Programming (CP2003)* (Sep. 2003)

[20] Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press, Cambridge MA (2004)

[21] Van Roy, P.: Ozma: Extending Scala with Oz Concurrency. Invited talk, QCon International Software Development Conference, San Francisco, CA, URL: `qconsf.com`, (Nov. 2011)