Ozma: Extending Scala with Oz Concurrency

Sébastien Doeraene

Abstract

Ozma is a conservative extension to the Scala programming language with Oz concurrency. Ozma adds dataflow values, light-weight threads, lazy execution and ports to Scala. The goals of this dissertation are to design and implement the language, its compiler and its runtime environment. The language should preserve Scala semantics as much as possible, ideally all. In order to achieve this, we have made decisions that limit the expressiveness of Ozma, compared to Oz. In particular, full unification is dropped, in favor of shallow unification (variable-variable). This dissertation describes the language, with a tutorial, examples and precise semantics, as well as the implementation of the compiler and the runtime environment.

Résumé

Ozma est une extension conservatrice du langage de programmation Scala, avec la concurrence de Oz. Ozma ajoute les valeurs dataflow, les threads légers, l'exécution paresseuse et les ports de Oz à Scala. L'objectif de ce mémoire est de concevoir et implémenter le langage, son compilateur et son environnement d'exécution. Le langage doit préserver le plus possible la sémantique de Scala, idéallement la conserver intacte. Pour cela, nous avons pris certaines décisions qui limitent l'expressivité de Ozma, comparée à celle de Oz. En particulier, l'unification totale est écartée, au profit d'une unification de surface (variable-variable). Ce mémoire décrit le langage, au moyen d'un tutoriel, d'exemples et d'une sémantique précise, ainsi que l'implémentation du compilateur et de l'environnement d'exécution.

I would like to thank:

- Pr Eric Steegmans, for his course at the Katholieke Universiteit Leuven, which made me discover the Scala language,
- Aleksandar Prokopec and Pr Charles Pecheur, for having accepted to read this master thesis,
- Stewart Mackenzie, for his interest in my project and making me tweet about it,
- Kevin Glynn, for his numerous comments, and reminding me all along that not all readers are Scala gurus,
- And Pr Peter Van Roy, for always asking for better and more ambitious results.

Contents

Introduction

Programming in concurrent, parallel, or distributed settings has become a necessity that no one denies anymore. Multicore CPUs, clusters, the Internet, and recently cloud computing are as many reasons to design with concurrency in mind. All languages of practical importance have their dedicated libraries and idioms to work in such settings. There are also languages that were designed for concurrency and distribution.

One of those languages is Oz, a multiparadigm programming language that provides advanced primitives regarding concurrency and distribution. It is mostly declarative, a paradigm that encompasses functional and logic programming. Despite its very innovative features and its expressiveness, Oz never made it into the wide developer community. Among the most likely reasons are its syntax, which is so different from all mainstream languages that it is difficult for developers to accept it.

There have already been works on bringing Oz ideas into better known languages.

Alice ML [Ros07] is an independent language, based on ML but running on a dedicated runtime system. Although theoretically and technically very interresting, it was not accepted by the community. The fact that its implementation is totally incompatible with any other existing tools makes it unusable in practice. Also, it started from a relatively unknown language, ML (compared to mainstream languages like Java or C#).

Flow Java [DSHB03] was designed as a conservative extension of Java with single assignment variables and futures. It was implemented using a modified JVM (Java Virtual Machine). Unfortunately, it was not accepted either. Java is an imperative language, which makes it difficult to fit in concepts from Oz. Besides, the Java community does not accept easily modifications to the JVM.

When designing the Scala programming language, its author, Martin Odersky, was very aware of the critical requirements for a language to be accepted by the community. He designed Scala as being both close to Java (in terms of syntax and concepts) and interoperable with any existing Java library. As of today, Scala seems to be the best hope of getting functional programming languages in mainstream languages.

Although Scala has no language feature related to concurrency, there exists an advanced library, Akka (http://akka.io/), created by Jonas Bonér, which provides Scala programmers with concurrent and distributed concepts coming from Erlang [Arm03]. It uses the Actor Model combined with Software Transactional Memory.

In 2011, Martin Odersky and Jonas Bonér, together with Paul Phillips (leading contributor to the Scala compiler), founded Typesafe, a company whose aim is "to create a modern software platform for the era of multicore hardware and cloud computing workloads."[1] This company develops open source Scala tools, and provides commercial support.

ScalaFlow is another library developed by Kai H. Meder in his master thesis [Med10]. It introduces dataflow variables from Oz in Scala at the library level. This allows to use them in

[1] http://typesafe.com/company

specific areas of the code, but dataflow variables cannot enter existing code designed for normal values.

Goals

At the heart of the motivation for this work, we would like to spread Oz concurrency ideas to mainstream communities. We believe that Scala is the perfect entry point for this. Scala is a functional language, which makes it rather close to Oz. At the same time, it is growing fast and getting interest from many people, including enterprises.

In order to achieve this, the main goal of this dissertation is to design and implement a conservative extension of Scala with the ideas of Oz, which we call Ozma.

Unlike ScalaFlow, we want dataflow variables to be integrated in the language, in order to be fully interoperable with existing Scala libraries. Unlike Flow Java and Alice ML which use a dedicated runtime, Ozma is designed to run on top of the existing implementation of Oz: Mozart[2].

Moreover, we want Ozma to preserve as much of Scala idioms while working in a concurrent setting. Both functional and object-oriented programs can become naturally concurrent in Ozma.

On the long run, we hope that features of Ozma will find their way to the official Scala language.

Contributions of this work

This master thesis brings the following contributions.

We have designed the Ozma language as a conservative extension of Scala with Oz concurrency. The semantics of Ozma are given in chapter 5.

We have implemented the parts of the compiler that are specific to Ozma semantics. We reused the official Scala compiler for all Scala semantics. We have also implemented a code runner, based on the Mozart runtime engine, and a small fraction of the Java runtime library for use by Ozma. The implementation is discussed in chapter 6.

Finally, we have studied the expressiveness of Ozma, with a tutorial and with examples (chapters 3 and 4 respectively). Examples are taken from [VH04] and from a programming course on Oz given at the Université Catholique de Louvain by Peter Van Roy. In particular, we reimplemented the end-of-term programming project of 2008 in Ozma (introduced in section 4.3.4), as a proof-of-concept that Ozma is expressive enough to cover the features of Oz taught in this course.

Structure

The first chapter introduces the Scala language and some of its features that are affected in Ozma. The second chapter provides the reader with shallow knowledge of Oz, highlighting the features that Ozma adds to Scala.

Chapters 3 to 5 describe the Ozma language and how to use it. Chapter three introduces the features of the language in a tutorial. Chapter four provides example programming techniques in this language. And chapter five gives the semantics of Ozma.

The sixth chapter discusses the implementation of Ozma: compiler, code runner and core classes.

[2] http://www.mozart-oz.org/

Chapter seven explores possible future developments.

Why the name Ozma?

Ozma takes its roots in two languages: Scala and Oz. As such, we have given it a name that is sound in both communities.

On the Oz side, traditionally, tools and libraries related to Oz are given names taken from the Land of Oz universe, created by L. Frank Baum. Ozma is the princess of the land of Oz.[3]

On the Scala side, it seems that there are a few names ending in 'a'. Among them are Java, Scala and Akka.

Software

This dissertation is accompanied by an implementation of the Ozma compiler, as well as examples of programs written in Ozma. Full source code and compilation instructions can be found at https://github.com/sjrd/ozma.

The printed version of this master thesis also includes a CD-ROM with the source code and pre-compiled software. Using this pre-compiled version, only the following instructions are needed.

In order to run the Ozma compiler and run example programs, the following software are required:

- Scala $\geq$ 2.9.0, available at http://www.scala-lang.org/downloads
- Mozart $\geq$ 1.4.0, available at http://www.mozart-oz.org/download/

Executables for Scala and Mozart must be available in the system path. It is also recommanded that the `bin/` directory or Ozma be made available. The following should run properly:

```
$ scala -version
Scala code runner version 2.9.0.final -- Copyright 2002-2011, LAMP/EPFL
$ oztool version
1.4.0
```

You can check that everything works fine by compiling and running the "Hello World!" example:

```
$ cd <ozma>/docs/examples/helloworld/
$ <ozma>/bin/ozmac helloworld/HelloWorld.scala
$ <ozma>/bin/ozma helloworld.HelloWorld
Hello world!
```

or, if the `bin/` directory of Ozma is available in the path:

```
$ ozmac helloworld/HelloWorld.scala
$ ozma helloworld.HelloWorld
Hello world!
```

[3] See http://en.wikipedia.org/wiki/Princess_Ozma

Chapter 1

Scala

Scala is a multiparadigm programming language that compiles to the Java Virtual Machine.[1] It supports functional programming as well as object-oriented programming. The language was created by Martin Odersky in 2001, its first public release was in 2003. A second, redesigned version was released in 2006. The current version of Scala is 2.9.0.1 and was released in May 2011.

The popularity of Scala has grown ever since it was created. At present, it is widely known and used by enterprises. According to Typesafe, Inc., Scala is used by over 100,000 developers.

As such, there is lots of introductory material on Scala available on the Internet, as well as books. A good entry point for documentation is the official website: `http://www.scala-lang.org/`. This chapter will not try to reinvent the wheel. It will only focus on core aspects needed for the understanding of the rest of this dissertation.

1.1 Introduction to Scala

Let us first begin with a very small introduction to what a Scala program looks like. Here is the unavoidable Hello World in Scala.

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

This code introduces an `object`, that can be considered as a singleton class. It defines a method `main` whose only argument is an array of strings. Its body is trivial and only displays the "Hello, world!" message on the standard output.

As a functional language, Scala encourages the use of immutable values. These are introduced by the keyword `val`. Variables, introduced by `var`, are only used when necessary. Additionaly, Scala makes extensive use of local type-inference in order to ease the burden of the programmer. Most types are inferred by the compiler, so that we do not need to write them.

Local values and variables need to be initialized at the point where they are declared. This ensures that we never try to use a value that has not yet been initialized.

This little code snippet computes the sum of two integers given on the command-line:

[1] Another implementation compiles towards .NET.

```
object Sum {
  def main(args: Array[String]) {
    val left = args(0).toInt
    val right = args(1).toInt
    println(left + right)
  }
}
```

In this example, we see that the type of `left` and `right` has been inferred by the compiler from the right-hand side of the equals sign. We used `val`'s because there is no need to modify `left` and `right`.

We can easily generalize this example to compute the sum of any number of command-line arguments. Thanks to the excellent collection framework of Scala and to the advanced local type inference, we can write it very concisely and elegantly.

```
object Sum {
  def main(args: Array[String]) {
    val elems = args map (_.toInt)
    println(elems.sum)
  }
}
```

The `map` function returns a collection of the same type as the source collection (here, an array), whose elements are the result of applying the given function to each source element. Here we map each source element, that is a string, into the corresponding integer.

The `sum` method simply computes the sum of the elements in the collection.

1.2 Working with lists

Arrays are mutable data structures. Hence, they are not that used in Scala, which is mostly functional. In Scala, we prefer to use immutable data structures wherever possible. A particularly important data structure in Scala are lists.

Lists are very common in functional languages. They even form the basic data structure of certain languages, like Lisp. In Scala, lists are not (totally) built in the language. They simply consist of instances of the class `List`. This class has exactly two subclasses: `Nil` (which is a singleton object) and `::` (pronounced cons).

`Nil` represents the empty list. An instance of `::` has a *head* and a *tail*. The head is the first element of the list, and the tail is the rest of the list, and is itself a list. We can use the infix, right-associative operator `::` to build instances of `::`, and hence of lists. For example, the list consisting of $3, 5, 1$, in this order, is written:

```
val list = 3 :: 5 :: 1 :: Nil
```

and is equivalent to

```
val list = ::(3, ::(5, ::(1, Nil)))
```

Scala offers a simpler way of writing a list:

```
val list = List(3, 5, 1)
```

We usually write recursive functions in order to work on lists. For example, the following function outputs the elements of a list:

```
def displayList(list: List[Any]) {
  if (!list.isEmpty) {
    println(list.head)
    displayList(list.tail)
  }
}
```

Another way of defining this function is by use of pattern matching.

```
def displayList(list: List[Any]) {
  list match {
    case Nil => ()
    case head :: tail =>
      println(head)
      displayList(tail)
  }
}
```

Actually, the collection framework allows us to write this function in a single line:

```
def displayList(list: List[Any]) = list foreach println
```

1.2.1 Tail recursion

Functional languages tend to use immutable variables and most loops are written as recursive functions. From a resource point of view, this is a problem, since working on a list with N elements uses a stack of N recursive calls.

Nearly all functional languages support some kind of *tail call optimization* to lift the problem. When a recursive call occurs at the last call of a function, it is transformed, either at compile time or at run time, into a jump to the beginning of the function. This ensures that, even if we write standard iterations as recursive calls, they execute with the same spatial complexity as imperative loops.

The `displayList` function that we showed previously is a good example: the recursive call is the last instruction in the body of the function. Scala has a dedicated phase in the compiler that transforms recursive calls into jumps. We can see that with the advanced option `-Xprint:tailcalls` of the Scala compiler. After tail call optimization, the method `displayList` has been rewritten as:

```
def displayList(list: List[Any]): Unit = {
  <synthetic> val _$this: Test.type = Test.this
  _displayList(_$this,list){ // this is a label definition
    if (!list.isEmpty)
      {
        println(list.head)
        _displayList(Test.this, list.tail) // this is a jump
      }
  }
}
```

It is good practice to always write tail-recursive functions when possible. Consider we want to write a function that concatenates two lists. A simple, but naive way to write it is:

```
def append[A](front: List[A], back: List[A]): List[A] = {
  if (front.isEmpty)
    back
```

```
  else
    front.head :: append(front.tail, back)
}
```

This function does work, but is not tail-recursive: there is a call to `::` after the recursive call. We can write a completely different implementation that *is* tail-recursive. It uses a helper function that reverses a list.

```
def reverse[A](list: List[A], acc: List[A] = Nil): List[A] = {
  if (list.isEmpty)
    acc
  else
    reverse(list.tail, list.head :: acc)
}

def append[A](front: List[A], back: List[A]): List[A] = {
  def loop(reversedFront: List[A], back: List[A]): List[A] = {
    if (reversedFront.isEmpty)
      back
    else
      loop(reversedFront.tail, reversedFront.head :: back)
  }

  loop(reverse(front), back)
}
```

The combination of these two methods is clearly more complicated, and they degrade readability. But they have the important advantage that they are tail-recursive, which is a desirable property.

1.3 Boxing and unboxing

In Scala, everything is an object, even primitive types. This makes it possible to parameterize generic types with primitive types, while keeping a single implementation (this is opposed to the way C++ uses templates, for example).[2]

This is conceptually true, but somewhat false in practice. Scala runs on the JVM, which does *not* consider a primitive type as an object. Since the early versions of Java, there have been *wrapper classes*: a class per primitive type which simply wraps a value of the corresponding type. This allows to store integers in a collection, for example. We simply wrap each integer in an instance of the `Integer` class, and put these instances in the collection. When retrieving elements from the collection, we unwrap it.

Starting from Java 5, the Java compiler has been doing *auto-boxing*, doing this automatically when required. If one tries to pass a primitive integer to a method that expects an `Integer` instance, or even an `Object`, the compiler automatically wraps it in an instance of `Integer`. It also *unboxes* instances of wrapper classes to their corresponding primitive types.

Scala also provides auto-boxing, but in a different way. As shown in figure 1.1, in Scala, there are three top-level classes: `Any`, `AnyVal` and `AnyRef`. Primitive values conceptually extend `AnyVal`, while all classes extend `AnyRef`. Both `AnyVal` and `AnyRef` extend `Any`.

Before *erasure* (see section A.3.5), the Scala compiler treats primitive types as being proper classes. Erasure is an important step in the compilation process that gets rid of generics, as well

[2]We assume familiarity with generic types in Java. If need would be, the reader can refer himself to http://download.oracle.com/javase/tutorial/java/generics/index.html.

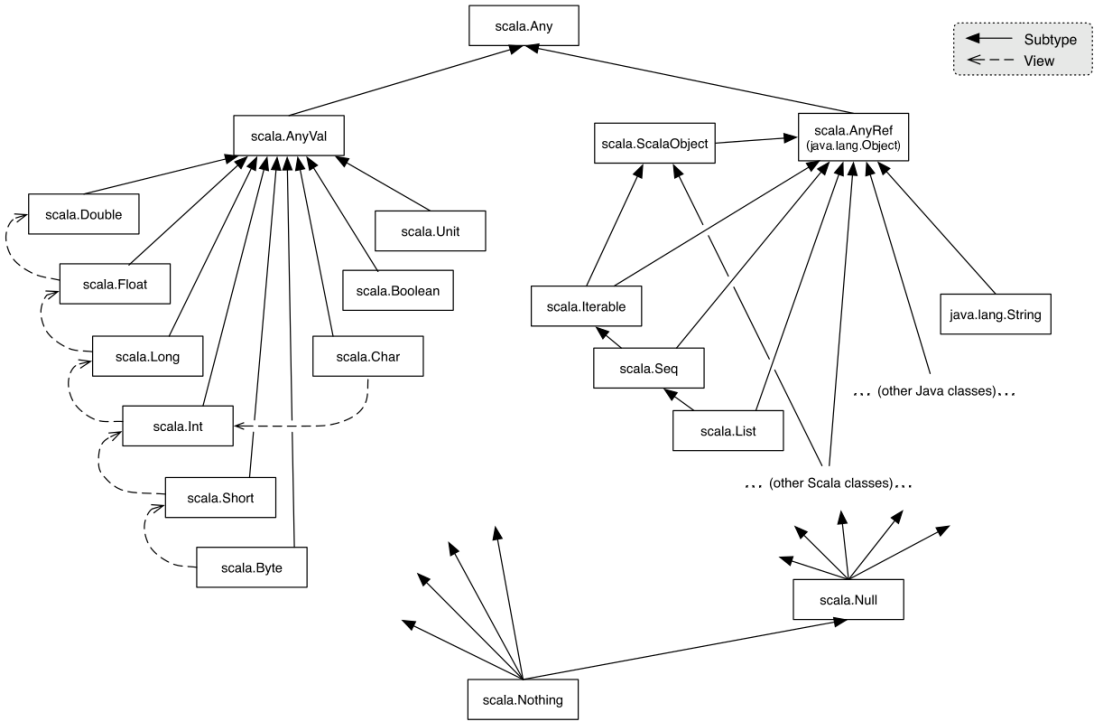


Figure 1.1: Scala type hierarchy [Ode11, fig. 12.1]

as `Any`, `AnyVal` and `AnyRef`. Values of generic types or of one of these three top-level types are all retyped to `Object`, which is the top-level class in Java.

Doing this, the relation between primitive types and `AnyVal` is broken, because an integer is *not* an `Object`. To compensate this, erasure also adds boxing and unboxing operations that fix this. Therefore, ultimately, putting an integer into a collection will be compiled as a boxing operation.

Let us put that in an example:

```
def main(args: Array[String]) {
  val x = args.length
  val option = Some(x) // Some[Int]
  if (!option.isEmpty) {
    val value = option.get
    println(value) // println(x: Any)
  }
}
```

This little code illustrates two causes of boxing: a generic type, and a value of type `Any`. Eventually, the Scala compiler will translate this code with erased types, giving:

```
def main(args: Array[String]) {
  val x = args.length
  val option = Some(scala.Int.box(x)) // boxing
  if (!option.isEmpty) {
    val value = scala.Int.unbox(option.get) // unboxing
```

```
    println(scala.Int.box(value)) // boxing
  }
}
```

Although boxing and unboxing are not part of the semantics of Scala [Ode11], it is important to be aware of them for two reasons. On the one hand, they introduce non-negligible performance overhead, as we need to create objects just to wrap values of primitive values. This is of sufficient importance that Scala provides user-directed *specialization* of generic classes and methods [Dra10]. On the other hand, the fact that boxing and unboxing operations are supposed to be transparent will impact the semantics of Ozma, as section 5.1.5 will show.

Chapter 2

Oz

Oz is a multi-paradigm programming language specialized for concurrent and distributed systems. Oz 1 was first released in 1995. The current version of the language is Oz 3.

Oz has very precise, mathematical semantics. It is defined as a set of computation models, each one with its properties and expressiveness. The models are structured in layers: an upper layer can express everything that a lower level can express, and adds a definite set of primitives.

The core computation model of Oz is declarative programming. The declarative core of Oz supports functional and logic programming. It is easily extended with concurrency without losing its good properties, giving declarative concurrent programming.

Our work is mainly focused on declarative concurrency, and the benefits it offers to programmers. We will also discuss another extension that adds state to programs: message-passing concurrency with ports.

Oz has other models that we will not present here, like constraint programming.

Compared to other languages, features of Oz related to concurrency and distributed systems are very powerful, as they are provided transparently by the language. Most of the time, we need not care about concurrency. We can write concurrent programs as easily as sequential programs.

A comprehensive introduction to Oz and its paradigms can be found in [VH04]. This book is the *de facto* reference on the language.

This chapter will provide the reader with a shallow knowledge of Oz, that should be sufficient to understand the rest of this dissertation.

2.1 Introduction to Oz

The syntax of Oz is very different from those of most mainstream languages. Let us begin with a "Hello world!".

```
{Browse 'Hello, world!'}
```

This calls the 1-argument procedure `Browse` with the parameter `'Hello, world!'`, which is an *atom* (similar to a `Symbol` in Scala).

This can be run in the interactive environment of Mozart, which is built on top of Emacs. This environment can be launched with the executable `oz` :

```
$ oz &
```

Once you have written the small snippet in the main buffer, use the menu Oz|Feed buffer (C-. C-b, Emacs notation) to execute the code. This will display a dedicated window called

the *browser*. The browser displays a line with `'Hello, world!'`. The browser is a nice tool for experimentation. It can display the contents of any Oz value (atoms, but also numbers, booleans, and complex data structures such as lists).

2.1.1 Variables

We can declare variables with the `declare` keyword, and bind values to them using the `=` operator.

```
declare X Y Z in
X = 5
Y = 10
Z = X + Y
{Browse Z}
```

This code snippet expectedly displays the number 15 in the browser.

Variables, as the name does not imply, are *immutable*. Attempt to rebind an existing variable will *fail*:

```
X = 6
```

```
%********************* static analysis error ******************
%**
%** equality constraint failed
%**
%** First value:        5
%** Second value:       6
%** Original assertion: 5 = 6
%** in file "Oz", line 7, column 4
%** ------------------ rejected (1 error)
```

The rationale behind the name *variable* is that in Oz, variables are *mathematical* variables. Once determined, their value cannot be changed.

Note that variables do not have a *type*, as in Scala. Oz is entirely *dynamically typed*. This means that any variable can hold any kind of value. [VH04, p. 104-106] discusses some advantages and drawbacks of dynamic versus static typing. [Bou03] was an attempt at defining a statically typed version of Oz, using a global type-inference algorithm.

2.1.2 Procedures and functions

Procedures are declared like variables, and introduced with the keyword `proc`. Here is a procedure that takes two parameters, and displays their sum.

```
declare
proc {DisplaySum X Y}
   {Browse X+Y}
end
```

In can then be used as

```
{DisplaySum 3 4}
```

Since all variables are dynamically typed, it is legal to write

```
{DisplaySum 3 'hello'}
```

but this will raise an exception at runtime.

Functions are defined by the keyword `fun`. The result of a function is its body, like in Scala.

```
declare
fun {Square X}
   X * X
end

{Browse {Square 4}} % displays 16
```

2.2 Bound and unbound variables

Let us get back a few steps and look at this code snippet.

```
declare X Y Z in
X = 5
Y = 10
Z = X + Y
{Browse Z}
```

In this code, we first declare three variables without giving them a value. Recall from section 1.1 that in Scala, we had to initialize values immediately upon declaration.

So what happens in Oz if we try to display a value that has not yet been initialized?

```
declare X in
{Browse X}
```

This *does* have a meaning. The browser will display a `_`, meaning that the variable it was asked to display is *unbound*. If later on we bind a value to this variable, the browser will automatically *update*. The statement

```
X = 5
```

will replace the `_` in the browser by the number 5.

How is it possible? Unbound variables are not quite uninitialized variables, as we understand them in other languages. It is actually possible to work with unbound variables: bind them to other variables, passing them as arguments to procedures, etc. The `=` operator is more powerful than assignment in other languages. When binding two unbound variables, they are actually *unified*.

This means that, from that point on, they refer to the *same* memory node. If later on one of the two variables gets bound to a value, the other variable will get the same value. This means that

```
declare X Y in
X = Y
Y = 5
{Browse X}
```

will display 5.

2.2.1 Functions as procedures

Actually, functions do not exist in Oz. They are mere syntactic sugar for defining a procedure with an additional, *output-mode* parameter. The `Square` function we saw earlier is actually translated by the compiler to this procedure:

```
proc {Square X ?Result}
   Result = X * X
end
```

When we write a call statement in a position where an expression is expected, it is interpreted as a call to a function. Hence, the compiler introduces an unbound variable, and calls the associated procedure with this unbound variable as last parameter. This value is then used instead of the function result. The call to `Browse` is thus translated to:

```
local R in
   {Square 4 R}
   {Browse R}
end
```

The `local` statement introduces one or more local variables (here, `R`). The variables are only visible in the body of the `local` statement.

2.2.2 Blocking on unbound variables

Some operations cannot perform their action when given unbound variables. For example, the arithmetic operations need to know the value of their operands in order to compute a result. The following code

```
declare X Y in
Y = X + 5
{Browse Y}
```

will *block*, because `X` is unbound and we need it to perform the addition.

In a sequential setting, blocking without notice is bad: we do not get an error message with the position of the error. But in a concurrent setting, this is sound, because another thread can bind `X` later. If that happens, the thread trying to perform the addition will resume and continue normally. We will get back to this idea in section 2.4.1.

2.3 Working with lists: benefits of unbound variables

Declarative programming uses only immutable variables. Hence, it will also use immutable data structures, and thus lists. In Oz, we build lists with the atom `nil` and *pairs* head/tail, in a way that is similar to Scala lists. The list with elements 3, 5, 1 is written

```
declare L in
L = 3|5|1|nil
```

The `|` operator is right-associative, as was `::` in Scala. There is syntactic sugar available to write lists:

```
declare
L = [3 5 1] % equivalent to 3|5|1|nil
```

We can browse this complex data structure directly. The browser can recognize lists and display them nicely.

```
{Browse L}
```

We use recursive functions with pattern matching to work with lists in Oz. The following procedure displays the elements of the list one by one in the browser. Compare the following function with the second definition of `displayList` in section 1.2.

```
declare
proc {DisplayList List}
   case List
   of Head|Tail then
      {Browse Head}
      {DisplayList Tail}
   [] nil then
      skip
   end
end
```

`skip` is the empty statement, i.e. it does nothing.

2.3.1 Lists and unbound variables

Lists and unbound variables make a very good pair. We can introduce *holes* in complex data structures, that are unbound variables. Consider this "unfinished" list:

```
declare Xs Ys in
Xs = 3|5|1|Ys
{Browse Xs}
```

The browser displays `3|5|1|_` (it may even be smart enough to display `Ys` instead of `_`). The tail of this list is actually unbound, while the rest of the list is determined. Since `Ys` is unified to this tail, binding `Ys` to a new list part will extend `Xs`:

```
declare Zs in % *not* Ys!
Ys = 2|6|Zs
```

After this code is fed to the compiler, the browser updates the list and displays `3|5|1|2|6|_`. We can close the list by binding a finite list to `Zs`:

```
Zs = nil
```

This incremental building of a list is very important in many Oz coding patterns. In particular, it is very useful for recursive functions on lists, as we will now see.

2.3.2 Tail recursion

Oz supports tail call optimization at runtime. This means that the previous declaration of `DisplayList` is not *compiled* as a loop, but *executes* in constant stack space nevertheless.

So again, it is good practice to write iterative computations in a tail-recursive way. But Oz can do better tail call optimization than Scala. Thanks to unbound variables and unification, Oz can rewrite functions that are tail-recursive *modulo cons* – i.e. where the only operation that takes place after the recursive call is the `|` operator – as plain tail-recursive procedures. Consider the naive `Append` method:

```
fun {Append Front Back}
   case Append
   of Head|Tail then
      Head|{Append Tail Back}
   [] nil then
      nil
   end
end
```

This function is tail-recursive modulo cons. The compiler rewrites such functions as procedures with an output parameter.

```
proc {Append Front Back ?Result}
   case Append
   of Head|Tail then
      local
         ResultTail % declares a local variable
      in
         Result = Head|ResultTail
         {Append Tail Back ResultTail}
      end
   [] nil then
      Result = nil
   end
end
```

This alternative rewriting makes use of two important properties we saw earlier: (a) functions are procedures with a result argument anyway and (b) we can build lists incrementally using initially unbound variables as their tails.

Therefore, the "naive" implementation of `Append` is actually perfectly good in Oz. Most other standard operations on lists can be written like that, such as `Map`, `Filter`, etc.

2.4 Oz concurrency

As was highlighted at the beginning of this chapter, Oz was designed for concurrency and distributed settings. This section will present three successive extensions of the declarative model that provide more and more expressiveness, related to concurrency: declarative concurrency, lazy execution and ports.

2.4.1 Declarative concurrency

The declarative concurrency model is a very simple, yet extremely powerful extension of the declarative model. It consists of a single new statement:

```
thread
   Statements
end
```

causes `Statements` to be executed in a new, light-weight thread. A light-weight thread is not ruled by the operating system, but by the runtime engine. The engine of Mozart can handle millions of simultaneously runnable threads.

The `thread` block can also be used as an expression:

```
declare
X = thread {SomeLongComputation} end
```

is a syntactic sugar for

```
declare X in
local Temp in
   thread
      Temp = {SomeLongComputation}
   end
```

```
   X = Temp
end
```

Because of the implicit blocking on unbound variables when they are actually needed, we can then write

```
declare
Y = X + 1
```

This will block until the execution of `SomeLongComputation` is finished, so that `X` gets bound to an actual value through its unification with `Temp`. Hence, introducing the thread has not changed the global effect of the program.

Dataflow variables

With the introduction of threads, the behavior of *waiting* for unbound variables to get bound, instead of crashing, becomes very useful. Indeed, the unbound variable can be bound by another thread. As soon as this thread binds the variable, the original operation can continue normally.

We call variables that are used in this way *dataflow variables*. This name highlights the fact that the control flow is mostly determined by the flow of *data*. A computer thread will eventually bind a value to a dataflow variable, and through the unification and waiting mechanism, this value will *flow* towards a consumer thread.

Consider this very naive implementation of Fibonacci's series:

```
declare
fun {Fibonacci N}
   if N =< 1 then
      1
   else
      {Fibonacci N-1} + {Fibonacci N-2}
   end
end

{Browse {Fibonacci 10}} % displays 89
```

This implementation follows the mathematical definition of $F(n)$. Its complexity is $O(2^n)$. This is a very bad implementation because there exists a simple algorithm that is $O(n)$, but that is not the point right now.

The point is that we can trivially make this computation multi-threaded. It is enough to wrap one of the recursive call with `thread`:

```
   else
      thread {Fibonacci N-1} end + {Fibonacci N-2}
```

This will compute the results of `{Fibonacci N-1}` and `{Fibonacci N-2}` in two separate threads. The spawned thread will be joined with the main thread through the implicit waiting introduced by the `+` operator.

Adding the `thread` operation does not change the global effect of the computation. It merely makes the program multi-threaded, virtually for free.

A more useful example is the parallel map on lists. The sequential version is

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

```
      nil
   end
end
```

It can be used like this:

```
fun {Sq X}
   {Delay 500}
   X*X
end

Xs = [1 2 3 4 5]
{Browse {Map Xs Sq}}
```

This will display `[1 4 9 16 25]` after 2.5 seconds. We can make a parallel version of `Map` with a simple `thread` expression:

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

Using this definition, we immediately get a list of 5 unbound elements. These 5 elements get bound to their respective values half a second later. The 5 executions of the `F` function were executed in parallel.

Incremental computation

Sections 2.3.1 and 2.3.2 showed that we often use unbound variables as the tail of a list. This is more important still with threads, as we can incrementally compute the transformation of a partial list.

Consider this code:

```
declare
fun {Gen From To}
   if From =< To then
      {Delay 200}
      From|{Gen From+1 To}
   else
      nil
   end
end

declare
Range = {Gen 1 10}
Result = {Map Range fun {$ X} X*X end}
{Browse Result}
```

This program displays the 10 first perfect squares in the browser. But we need to wait until `Gen` has finished its work before getting any result. If we simply wrap the calls to `Gen` and `Map` in threads, we can see the partial results as they are computed.

```
declare
```

```
Range = thread {Gen 1 10} end
Result = thread {Map Range fun {$ X} X*X end} end
{Browse Result}
```

It is as simple as that. This works because of two important properties of Oz:

- Functions that are tail-recursive modulo cons are made tail-recursive, thus yielding partial results when given partial inputs.
- The successive tails of the lists act as *dataflow variables*, making communication between threads completely transparent.

When we use lists and threads this way, we speak of *streams*. A stream is nothing more than a list whose tail is temporarily unbound, and is shared by a producer (`Gen`) and a consumer (`Map`).

We will not elaborate on stream programming techniques now. This will be deferred until section 3.3.3, and done directly in Ozma.

2.4.2 Lazy execution (demand-driven execution)

Threads and dataflow variables are very powerful, but they are not enough, because they impose a supply-driven control flow. The producer generates as many elements as it can in the stream, and the consumer application is triggered by the availability of items in the stream.

Oz also supports demand-driven execution, through the primitive `WaitNeeded`. `WaitNeeded(X)` suspends the current thread until `X` becomes *needed*. This allows us to delay the computation of a value until this value is actually needed by its consumer.

Consider this declaration:

```
declare X in
thread
   {WaitNeeded X}
   {Browse 'start computation'}
   X = 5
end

{Browse X}
```

If we feed this to the emulator, the browser displays a `_` (or `X`). The atom `'start computation'` is not displayed, because the thread is suspending on `WaitNeeded`. If we make `X` needed by issuing this line,

```
{Browse X+1}
```

then the thread automatically resumes, and binds `X` to 5, which, in turn, resumes the main thread that displays 6 in the browser.

Oz provides higher abstractions built on top of `WaitNeeded`: the standard functions `ByNeed` and `ByNeedFuture`, as well as syntactic sugar to declare *lazy* functions. We will not discuss `ByNeed` and `ByNeedFuture` here, as we will present them directly in Ozma in section 3.3.5.

The `lazy` annotation allows to declare very easily a lazy function:

```
declare
fun lazy {LazyFun X}
   {Browse 'in LazyFun'}
   X*X
end
```

```
Y = {LazyFun 4}
{Browse Y}
```

This only displays a _ (or `Y`) in the browser. The atom `'in LazyFun'` is not displayed, because the body of the function has not been executed yet. If we make `Y` needed,

```
{Browse Y+1}
```

then the execution of the function is triggered. In fact, the `lazy` annotation has transformed the function as it were declared like this:

```
proc {LazyFun X ?R}
   thread
      {WaitNeeded R}
      {Browse 'in LazyFun'}
      R = X*X
   end
end
```

2.4.3 Ports

The last computation model of Oz that we will discuss in this introduction is the message-passing model with ports. A port is simply the consumer of a stream, but which can deal with multiple independent producers. We create a port using the primitive `NewPort`:

```
declare Xs P in
{NewPort Xs P}
{Browse P}
{Browse Xs}
```

This binds `P` to the port handle, and `Xs` to a so-called *future*. Roughly speaking, a future is a read-only view of an unbound variable.

We can use the port handle to *send* data to the port:

```
{Send P 1}
{Send P 10}
```

This will append the elements 1 and 10 (in this order) at the end of `Xs`. The browser will automatically update to reflect the changes.

Ports cannot be implemented in the previous models, they need to be a primitive. This is because the declarative models are deterministic. Ports introduce nondeterminism in the program. We can highlight the nondeterminism using several producer threads:

```
thread {Send P 1} end
thread {Send P 2} end
thread {Send P 3} end
```

Depending on the interleaving of the three threads, this can append to the port stream any permutation of $(1, 2, 3)$.

We almost always use ports in conjunction with a consumer thread, that iterates over the elements of the stream:

```
declare
proc {Consumer Xs}
   case Xs of X|Xr then
      % do something with X, e.g.
```

```
      {Browse X}
      {Consumer Xr}
   end
end

declare P in
thread
   local Xs in
      {NewPort Xs P}
      {Consumer Xs}
   end
end

thread {Send P 1} end
thread {Send P 2} end
thread {Send P 3} end
```

We will get back at port programming techniques in the following chapter, in section 3.3.6.

Chapter 3

Ozma

Ozma is the language designed in this master thesis. It extends Scala with Oz concurrency features. This chapter is devoted to its description. First, we shortly discuss the macro design decisions. Then, we introduce the language *in a nutshell*. Finally, we provide a tutorial describing its features in depth.

The next chapter provides examples of common programming techniques in Ozma.

3.1 Design issues

3.1.1 Scala libraries

Although the Scala has no *language* feature related to concurrency, it has an important standard library which has a well defined API for working in concurrent settings: actors, parallel collections, the Akka framework, etc. Since Ozma extends Scala with concurrent features, it makes sense to decide whether or not to modify the existing concurrency model of Scala. We will explore both solutions and discuss their respective benefits and disadvantages.

In this dissertation, we settled on the choice to keep the existing libraries as is, while adding conservatively specific concepts in Ozma, and providing dedicated libraries. The implications of modifying the existing libraries were out of the scope of this work.

The first option is to rethink completely the concurrency model of Scala. That would mean that parallel collections, or futures, would be overwritten in Ozma.

Doing so, we could substantially improve the implementation, and use cases, of the Scala standard concurrency libraries. That would globally improve applications using these APIs. For example, parallel collections can be implemented trivially with light-weight threads (see section 3.3.3). Futures are an even better example, since dataflow values encompass their usage in a language-integrated way.

On the down side, we would have to be careful not to alter the specifications of existing APIs. This could prevent us from possible improvements. In Scala, streams are lazy, while in Oz they can be either lazy, eager, or a mix of both (bounded buffer). We could be tempted to extend the `Stream` class with these use cases, but that would likely break existing applications relying on the fact that streams are always lazy.

The other option is to keep the existing Scala libraries as is, and add a new one for Ozma, with specific use cases that take advantage of the features of Ozma.

Using this approach, we do not do anything to improve existing applications. And we take the risk of offering two completely different APIs to programmers. This could lead to incompatibilities between certain applications.

However, we can take full advantage of all the concurrency model of Oz, which is quite nice, as we saw in Chapter 2. We do not need to hold tight to existing APIs.

As we already said, we will continue this work using this design.

3.1.2 Unification

Oz supports full unification. That means that the = operator will dive into data structures, and make them *structurally* equal. In Ozma, structures do not exist: all data structures are objects. Now, the = operator does not dive into objects.

Therefore, Ozma does not support full unification. It only supports variable-variable, i.e. *shallow* unification. This means that we cannot unify two partially filled lists and expect them to merge: this will fail instead because the two lists are two different instances.

Shallow unification is *essential* to all Oz programming techniques, while full unification is only rarely needed. We chose to drop full unification in Ozma because it fits better in the object model, while still preserving most of Oz programming techniques.

Note that this shallow unification is still very powerful: many unifications can take place in any order, and the result will always be the same.

3.2 Ozma concurrency extensions in a nutshell

3.2.1 Tail call optimization of list functions

Methods that are tail-recursive modulo cons, i.e. whose only operation following the recursive call is a list cons (::), are compiled as loops, as if they were tail-recursive. For example, the following `append` method is tail call optimized:

```
def append[A](front: List[A], back: List[A]): List[A] =
  if (front.isEmpty) back
  else front.head :: append(front.tail, back)
}
```

This property eases considerably the development of list processing functions.

3.2.2 Light-weight threads

In Ozma, we can spawn light-weight threads very easily, using the primitive `thread`:

```
println("Main thread")
thread {
  println("New light-weight thread")
}
println("Continuing main thread")
```

3.2.3 Dataflow concurrency

The main extension to Scala, in Ozma, is *dataflow concurrency*. In Ozma, every value (`val`) is a dataflow value, like every variable is a dataflow variable in Oz. Values can be unified with each

other before they are given an actual value. When a value gets bound, all other values unified with that one also get bound to the same actual value.

Dataflow values are mostly used in concurrent programs to achieve transparent data-driven communication between threads. A thread that needs a value that has not been bound yet will *wait* until it becomes bound. Programs using dataflow concurrency are always *deterministic*, which is a very nice property for concurrent programs.

The following code always behaves the same, i.e. prints 3 on the console. The third thread will wait until both `x` and `y` are bound, and the `println` call will wait until `z` is bound.

```
val x: Int
val y: Int
val z: Int

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

println(z)
```

3.2.4 Using thread as an expression

It is possible to use the `thread` primitive as an expression. In that case, the compiler implicitly declares a dataflow value, that will be bound to the result of the threaded expression when it is done. Using this property, the previous example can be rewritten as:

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

3.2.5 Streams: lists as dataflow communication channels

Dataflow values are everywhere, which means that all standard classes and methods use dataflow concurrency. In particular, lists are often used as communication channels between threads:

```
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
```

3.2.6 Lazy execution

The `byNeed` standard function introduces a *lazy* dataflow value. It will be evaluated on-demand. It is most useful in combination with streams. Lists standard methods can be made lazy by prefixing them by `.lazified`:

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

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

3.2.7 Message-passing concurrency

All the previous features live in the *deterministic* subset of Ozma. For nondeterministic needs, Ozma provides *message-passing* concurrency by means of ports. A port is a FIFO list of messages that can accept input from multiple threads.

```
val port = Port.newStatelessPortObject { message: Int =>
  println(3*message)
}

thread { for (i <- 1 to 10) port.send(i) }
thread { for (i <- 11 to 20) port.send(i) }
```

3.3 Tutorial

This section introduces the features of Ozma in a tutorial form. This will bring insight to the reader on what Ozma is good for. Precise semantics of the concepts introduced here will be detailed in chapter 5.

3.3.1 Bound and unbound values

In Scala, all local values must be assigned a value upon declaration, like in

```
val x = 5
val y: Int = 5
```

In the first declaration, the type of `x` is inferred by the compiler from the type of the expression on the right-hand side (rhs) of the `=`.

The following is illegal in Scala:

```
val x
val y: Int
```

This ensures that no value is unassigned when we use it in an expression.

In Ozma, as in Oz, we can declare a value without initializing it, provided we specify its type (since there is no rhs to infer the type from). So the following statement is legal:

```
val y: Int
```

This declaration introduces an *unbound* value.[1] Such a value is left uninitialized, until it gets bound to a value through the use of a *binding* operation:

```
y = 5
```

This looks like `y` is a variable and we *assign* a value to it, but it is not. If later on we try to bind another value to `y`, an Oz failure will be raised. That is to say, an unbound value acquires *single assignment semantics.*

[1] Be careful: in Oz we speak about unbound *variables*, and about *cells*, whereas in Ozma we have unbound *values*, as well as *variables*, respectively.

Passing unbound values around

Unbound values can be assigned or bound to other values and variables, even when they are not yet bound to an actual value. When it gets bound, all values and variables that were bound to this unbound value acquire the same value.

```
val x: Int
val y = x
x = 5
println(y) // displays 5
```

An unbound value can be inserted into a bigger data structure, e.g. a tuple. When the value gets bound, the internal field of the data structure is also updated.

```
val x: Int
val t = (x, 5)
x = 10
println(t) // displays (10,5)
```

Such bindings remain valid across method boundaries. So that we can write:

```
def makeSomeTuple(x: Int) = (x, x)

val x: Int
val t = makeSomeTuple(x)
x = 10
println(t) // displays (10,10)
```

Blocking on an unbound value

As long as we bind unbound values to other values, the execution can continue. But some operations cannot be performed on an unbound value. If such a situation arises, the execution blocks:

```
val x: Int
println(x) // blocks and never displays anything
```

The `println` method blocks because it needs to know the actual value of `x` in order to convert it into a string. The precise list of operations that block when given unbound values is presented in section 5.1.4. For now, you can roughly consider the following:

- Arithmetic and logic operations, and comparisons,
- Calling a method on an unbound object (`x.doSomething` with `x` unbound).

In particular, boxing and unboxing operations do not block. They continue with an unbound boxed or unboxed value when given an unbound value (see section 5.1.5).

3.3.2 List functions are tail-recursive

As we already saw in the two previous chapters, lists play an important role in programming in both Oz and Scala. So obviously, they are also important in Ozma. Here is the same rudimentary example we saw in section 1.2.

```
val list = List(3, 5, 1)
displayList(list)

def displayList(list: List[Any]) {
  if (!list.isEmpty) {
    println(list.head)
    displayList(list.tail)
  }
}
```

Using the `foreach` method, this can be written as

```
val list = List(3, 5, 1)
list foreach println
```

The good news about lists in Ozma is that they share the good properties of Oz lists. Specifically, the `::` operator is tail-recursive. This means that we can implement a tail-recursive `append()` method very easily:

```
def append[A](front: List[A], back: List[A]): List[A] = front match {
  case Nil => back
  case head :: tail => head :: append(tail, back)
}
```

This `append()` method is tail-recursive modulo cons, which in Ozma is effectively compiled as a tail call. Recall from section 2.3 that the following function definition:

```
fun {Append Front Back}
   case Front
   of Head|Tail then
      Head|{Append Tail Back}
   [] nil then
      Back
   end
end
```

was transformed by the compiler as

```
proc {Append Front Back ?Result}
   case Front
   of Head|Tail then
      local
         ResultTail
      in
         Result = Head|ResultTail
         {Append Tail Back ResultTail}
      end
   [] nil then
      Result = Back
   end
end
```

The Ozma compiler does exactly the same transformation. The application of the `::` method is inlined as a call to the constructor of the `::` class. And it knows that the second argument of this constructor can be extracted in tail call position. The main difference is that there is no way we can actually write the resulting code in Ozma anymore, because all parameters in Ozma are *input* parameters.

Nevertheless, assuming there was an `out` keyword that allowed a parameter to be an output parameter, we could write the `append()` method like this:

```
def append[A](front: List[A], back: List[A], out result: List[A]) {
  front match {
    case Nil => result = back
    case head :: tail =>
      val resultTail: List[A]
      result = ::(head, resultTail)
      append(tail, back, resultTail)
  }
}
```

Section 5.6 will show that the tail call optimization of Ozma is actually more general than that, because it is not restrained to lists. One can give tail call semantics to any parameter of any user-defined method, using the `@tailcall` annotation.

Tail-recursion modulo cons is an *essential* property of Ozma, as in Oz, because it allows to turn any function working on a list into a concurrent agent, without memory leak.

3.3.3 Declarative concurrency

The true power of Ozma is related to its concurrency model. In the previous chapter, we described three levels of concurrency that are specific to Oz. We will now introduce them in Ozma.

Recall from section 2.4.1 that declarative concurrency was introduced by a single primitive, i.e. the `thread` statement. In Ozma there is a builtin library method `thread` in the package `scala.ozma`. We can use it to spawn light-weight threads as easily as in Oz.

```
import scala.ozma._

thread {
  println("This is executed in a new thread")
}
println("This continues in the main thread")
```

A thread can also return a value, as in Oz:

```
val x = thread {
  someLongComputation()
}
println(x)
```

And, because in Scala braces and parentheses are (mostly) interchangeable, this can also be written as:

```
val x = thread(someLongComputation())
println(x)
```

Given that the body of the thread has type T, this can be understood as a syntactic sugar for

```
val x: T
thread {
  x = someLongComputation()
}
println(x)
```

Note that, in this last example, `x` truly becomes a dataflow value (called dataflow variable in Oz). Indeed, it is an initially unbound value, that therefore acquires single-assignment semantics. The `println` call blocks because it needs the actual value of `x`, which is unbound. But when the thread finishes, `x` becomes bound, and the `println` call can unblock and display its value.

Incremental computation

We saw that in Oz, we often use threads with lists whose tail is temporarily unbound, in order to compute things incrementally. Consider this short example:

```
import scala.ozma._

object Test {
  def main(args: Array[String]) {
    val range = gen(1, 10)
    val result = range map (x => x*x)
    result foreach println
  }

  def gen(from: Int, to: Int): List[Int] = {
    sleep(100)
    if (from > to)
      Nil
    else
      from :: gen(from+1, to) // remember, this is tail-recursive
  }
}
```

The call `sleep(100)` suspends the current thread for 100 ms.

When this example is run, it displays nothing for a second, then displays the entire result at once. This is a pity. We would like it to display incremental results as soon as some parts get computed. We can do that easily by introducing two simple `thread` calls:

```
  def main(args: Array[String]) {
    val range = thread(gen(1, 10))
    val result = thread(range map (x => x*x))
    result foreach println
  }
```

Now, a run displays each perfect square number in turn, with a tenth of a second between each. This works not only because of the threads, but also because the `::` operator is tail-recursive, meaning that the `gen` method (as well as the standard `map()` method) actually produces a partial result before it is completed. The combination of these two properties make lists behave as streams.

If you are not convinced that tail-recursion of `::` is needed, try replacing it by a non-tail-recursive variant:

```
def cons[A](head: A, tail: List[A]) = head :: tail
...
    else
      cons(from, gen(from+1, to))
```

Streams and declarative agents

With threads and incremental computation of lists, we can write a large number of applications based on streams. A stream is simply a list whose tail is unbound.

As an interesting example of computing with streams, we will write an application that generates prime numbers using the Sieve of Eratosthenes. This example is taken from [VH04, p. 260] and translated in Ozma.

The Sieve of Eratosthenes generates consecutive numbers starting from 2. It then filters out numbers that are multiples of already found primes. Consider the list consisting of the numbers 2, 3, 4, 5, 6, 7, 8, 9. The first number is 2 and is a prime number. The sieve yields it, then filters out all other numbers that are multiples of 2. At that point, 3, 5, 7, 9 are left. Then it yields 3, and filters out 9, thus leaving 5, 7. It will then successively yield 5 and 7.

This can be modeled as a producer-filter*-consumer. The producer generates all integers starting from 2 up to a given number. The consumer displays the list that it receives (this is `foreach println`). In between, we inject filters that will get rid of non primes. We will create one filter for each prime number that we encounter.

The producer, the consumer, and all the filters are run in separate threads. These threads communicate between each other using the streams. As such, each thread can be thought of as an *agent*, i.e. an active entity processing some input and producing some output.

```
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 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 filter (_ % head != 0))
        head :: sieve(filtered)
    }
  }
}
```

This program will display all prime numbers that are less than or equal to a given number N. This number should be given on the command-line, like this:

```
$ ozma PrimeNumbers 100
```

3.3.4 Memory management

Lists used as streams tend to be big. Often, they even have infinite size. It is therefore important to take memory management into account when working with streams.

Consider the following little program. It shows a basic producer-consumer pattern.

```
import ozma._
import scala.ozma._

object Summer {
  // Create a producer and a consumer thread, and display the result
  def main(args: Array[String]) {
    val high = if (args.length > 0) args(0).toInt else 150000
    val list = thread(generate(0, high)) // producer thread
    val result = thread(sum(list))       // consumer thread
    println(result)
  }

  // Generate a stream of integers between 'from' and 'to' included
  def generate(from: Int, to: Int): List[Int] = {
    if (from <= to) from :: generate(from+1, to)
    else Nil
  }

  // Compute the sum of a list
  def sum(list: List[Int], acc: Int = 0): Int = {
    if (list.isEmpty) acc
    else sum(list.tail, acc + list.head)
  }
}
```

The `main` method creates a thread for the producer, and another one for the consumer. The stream `list` makes up the communication channel between these two agents.

While this example is working, it is perfectible. The `sum` method is implemented using a classical accumulator pattern. This structure is provided by the `foldLeft` method of `List`. So this program can be rewritten a lot more concisely as follows:

```
import ozma._
import scala.ozma._

object Summer {
  def main(args: Array[String]) {
    val high = if (args.length > 0) args(0).toInt else 150000
    val list = thread(generate(0, high))          // producer thread
    val result = thread(list.foldLeft(0)(_ + _)) // consumer thread
    println(result)
  }

  def generate(from: Int, to: Int): List[Int] = {
    if (from <= to) from :: generate(from+1, to)
    else Nil
  }
}
```

However, this introduces a subtle memory leak, because we call `foldLeft` directly on the front of the stream. Despite the fact that `foldLeft` is internally tail-recursive, the list it is applied to cannot be reclaimed until the last recursive call terminates (i.e., never). More generally, the garbage collector cannot reclaim the `this` of a method ending in a tail call.[2]

[2]This seems to be an implementation limitation of the garbage collector, as it is clear that there is no more reference to the `this`. It might be improved in the future.

Since the stream can become very big, and its front cannot be garbage collected, the program cannot execute in constant space. Hence it has a memory leak, and will run out of memory if given a higher upperbound. On a computer running Linux 64 bits with 4-GB of RAM, the program crashes if asked to compute the sum of 3,000,000 integers.

Therefore, we cannot call methods that must act as agents directly on the stream. We must convert the stream to an agent view using the method `toAgent`:[3]

```
val result = thread(list.toAgent.foldLeft(0)(_ + _)) // consumer thread
```

Methods invoked on the agent view of a stream will release the reference to the front of the stream, so that it can be garbage collected (if not accessible elsewhere, of course).

As a final bonus in this section, let us mention the method `sum` in Scala collections, which is exactly what we need here. So, actually `foldLeft` is not even necessary:

```
val result = thread(list.toAgent.sum) // consumer thread
```

Fixing the Sieve of Eratosthenes

In the light of this section, we fix our implementation of the Sieve of Eratosthenes so that it manages memory correctly.

```
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      // toAgent here
  }

  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) // toAgent here
        }
        head :: sieve(filtered)
    }
  }
}
```

3.3.5 Lazy execution

So we have a program, written in the declarative concurrency model, that computes the prime numbers smaller than a given N. What if we would like to produce the N first prime numbers instead? Since it is not practical to determine a priori how many integers we need to generate in

[3]Implementation of the agent view is out of the scope of this discussion. The interrested reader can find it in `src/library/scala/ozma/ListAgent.scala`.

the first place, we simply generate them all. We will simply modify the consumer to only display the N first elements of the list it receives. This is trivially implemented using the `take` method of lists.

```
import scala.ozma._

object Test {
  def main(args: Array[String]) {
    val count = java.lang.Integer.parseInt(args(0))
    val integers = thread(generateFrom(2))
    val result = thread(sieve(integers))
    thread(result take count) foreach println
  }

  def generateFrom(from: Int): List[Int] = {
    from :: generateFrom(from + 1)
  }

  def sieve(list: List[Int]): List[Int] = // unchanged
}
```

This implementation actually works, because threads will communicate their streams. So the consumer does not have to wait for the never-happening termination of the producer. However, it is overwhelmingly inefficient. There is a good chance that the generator will get ahead of the filters (it is much faster), and will thus produce many numbers that will never be used.

The problem is not related to the fact that we generate infinitely many integers in the first place. The actual issue is that we let the generator produce elements faster than the consumer and filters can consume them. In order to fix this issue, we would like the producer to compute only elements that are needed by the consumer.

This kind of resource management can be achieved with the demand-driven computation model that we introduce in this section.

Wait until a value is needed

In order to support demand-driven computation, we add a new primitive to the language, `waitNeeded(x)`. Calling this method will suspend the current thread until the value referred to by `x` becomes *needed*. Roughly speaking, a value becomes needed when one of the following events occurs:

- Another thread blocks on `x` because it is unbound or
- `x` becomes bound.

At that moment, the thread that called `waitNeeded` will resume.

Let show this by an example:

```
val x: Int
thread {
  waitNeeded(x)
  println("Starting computation")
  x = someComputation()
}
println("Main thread continues")
sleep(1000)
```

```
println("Now we need x to display it")
println(x)
```

Running this example yields:

```
Main thread continues
[ 1 second elapses ]
Now we need x to display it
Starting computation
10
```

This shows that execution of the actual body of the thread was triggered by the last `println` call. Indeed, this call blocks on `x`. Hence, it makes `x` needed, which unblocks the call to `waitNeeded(x)` in the thread.

Using this new primitive, we can build bigger abstractions. In particular, the `byNeed` operation.

By-need execution

Often, we want to define a value as the result of some computation, but want to trigger the computation only when the value is actually needed. We already implemented something like this in the previous example: the call to `someComputation()` was triggered when the `println` call needed `x`. This is called *by-need execution*. We introduce a new abstraction, `byNeed`, that takes a parameter passed by name, and returns immediately with an unbound value. When this value is needed somewhere, the execution of the body is triggered, and bound to the value.

In the actual implementation of Ozma that will be presented in chapter 6, `byNeed` is a native method. But this is only for efficiency considerations. Actually we can implement it in Ozma using the concepts that were already introduced: dataflow values, threads, and `waitNeeded`.

```
def byNeed[A](value: => A) = {
  val result: A
  thread {
    waitNeeded(result)
    result = value
  }
  result
}
```

Using this abstraction, we can rewrite the example from the previous section more concisely and more elegantly:

```
val x = byNeed {
  println("Starting computation")
  someComputation()
}
println("Main thread continues")
sleep(1000)
println("Now we need x to display it")
println(x)
```

This program snippet yields exactly the same result as the one in the previous section.

Scala has also the concept of `lazy val`. The previous example can be written with a `lazy val` without change.

```
lazy val x = {
  println("Starting computation")
  someComputation()
}
```

However, lazy values are evaluated the first time they are *accessed* (i.e. encountered in the code). An Ozma by-need value is a dataflow value that can be passed around without evaluating it. Only when it becomes *needed* does it get evaluated. The difference is illustrated in this code snippet:

```
val x = byNeed { ... }
val y = x // x is not evaluated
println("checkpoint")
println(y) // now x is evaluated

lazy val x = { ... }
val y = x // x is evaluated here
println("checkpoint")
println(y)
```

Passing exceptions through

There is still a small issue with the `byNeed` abstraction. What happens if the computation in `byNeed` throws an exception?

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

This fails miserably to catch the exception correctly. Indeed, the exception was raised in another thread, where there is no surrounding exception handler. We would like the exception to be somehow rethrown in the main thread.

But where, and when, should the exception be thrown? We cannot throw it at the point where we call `byNeed(list.head)`, since we would need to wait for the termination of the computation at that point, which is in total contradiction with the purpose of `byNeed`. The only reasonable choice is to raise the exception at the point where `x` is made needed, i.e. at the `println` call.

Since that point could be anywhere else in the program, where we have no idea that `x` was in fact created as a lazy value, we must encode in the value itself the fact that its computation has failed. This concept exists in Oz, and is called a *failed value*. We can create a failed value wrapping an exception. When that value is needed, the wrapped exception is thrown.

In order to support this, we introduce a new primitive, `makeFailedValue(th: Throwable)`, that wraps the given throwable in a failed value and returns it. Any thread that blocks on that value will then throw the exception `th`.

With this primitive, we can write an alternative to `byNeed`, that we call `byNeedFuture`, that catches any exception thrown by the lazy computation, and wraps it in a failed value that is bound to the resulting value. Again, this method is actually native in the implementation for efficiency.

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

Rewriting our previous example and replacing `byNeed` by `byNeedFuture`, we get the desired behavior, namely that the string "The list was empty" is printed on the console.

The Sieve of Eratosthenes revisited

The reader might have been wondering when (and whether) we were going to get back at our initial problem, the Sieve of Eratosthenes. We finally have the truly lazy execution abstraction, which is `byNeedFuture`.

We introduce a method `lazified` applicable on `List`, which gives a lazy view of the list. Under the lazy view, the methods `map`, `filter`, `filterNot`, `take` and `drop` are lazy, using `byNeedFuture`.

Rewriting the naive implementation of section 3.3.5 using `byNeedFuture` and lazy views of lists instead of threads gives the following implementation:

```
import scala.ozma._

object PrimeNumbers {
  def main(args: Array[String]) {
    val count = args(0).toInt
    val result = sieve(generateFrom(2))
    (result.lazified take count) foreach println
  }

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

  def sieve(list: List[Int]): List[Int] = byNeedFuture {
    list match {
      case Nil => Nil
      case head :: tail =>
        head :: sieve(tail.lazified filter (_ % head != 0))
    }
  }
}
```

This program is part of the examples in the source code. It can be found in the directory `docs/examples/primes/`.

Thanks to lazy execution, the infinite generator will actually produce only those numbers that will be needed, eventually, by the consumer. Hence, there is no computational power lost for nothing, and we get better performance.

Note also that the lazified view of a list provides correct memory management, as does an agent view. There is therefore no need to use `toAgent` in addition to `lazified`.

3.3.6 Ports

With dataflow values, threads, and lazy execution, we can write a lot of concurrent programs. But all these abstractions are part of the declarative concurrent model. This model is useful for many problems, but is lacking *state*.

Ports introduce some kind of state in the model. They are thus not part of the declarative concurrent model. Ports are an easy way to express message-passing programs.

This section presents the message-passing concurrent model, which is an extension of the declarative concurrent model with ports.

Multiple writers in a stream

A port is essentially the consumer of a stream. The difference is that it can handle multiple producers for its stream. In the declarative concurrent model, only one thread can be the producer of a stream. Having multiple writers introduces nondeterminism in the program (a consumer cannot know which producer the next value will come from), and hence cannot be expressed in the declarative model.

We create a port using the method `Port.make`. We have to pass it a handler function that is the actual consumer function. Then we can `send` messages to the port from multiple producer threads. Each received message will be appended at the end of the stream. Let us see a trivial example:

```
val port = Port.make(displayList)

port.send(1)
port.send(2)
```

This creates a port that will display all elements that it receives. Note that the handler function is started in a new thread, and is initially given an unbound list.

We then send successively two "messages" (here, mere integers) to the port. They will be appended at the end of the list that was given to the handler, `displayList`. So, they will be printed on the console.

So we have one producer here. What happens if we make three producers:

```
thread { port.send(1) }
thread { port.send(2) }
thread { port.send(3) }
```

What will be the output? The answer is: we do not know. It can be any permutation of $(1, 2, 3)$. The program has become nondeterministic, because we have more than one producer thread for a single consumer. This is possible because of the port abstraction.

Agents with ports

Ports are often used to represent *agents*. An agent is an active entity that processes messages. It can have an internal state, or not.

With raw ports, we have to provide a handler function that works on a list. This list grows as new messages are sent to the port. More than often, we want to define a handler that processes a single message. The handler will then be called for each element of the list.

Obviously, calling a method for each element of a list is what the `foreach` method of class `List` does. In the previous example, we used `displayList` as an agent that displayed the entire list. Actually, we can view it as an agent that displays each message it receives, independently of the others. In this respect, the actual handler method is `println`. We can use an anonymous function calling `foreach` instead.

```
val port = Port.make((_:List[Any]).toAgent foreach println)
```

This removes the burden of defining `displayList` separately. This use case is so common that we want a higher abstraction for this. We call this abstraction a port object. The following method is defined in `Port`:

```
def newStatelessPortObject[A, U](handler: A => U) =
  make[A](_.toAgent foreach handler)
```

Using this abstraction, it is easier to define our display port:

```
val port = Port.newStatelessPortObject(println)
```

We can extend the concept of port object with implicit state. Consider that we want to improve our little display agent so that it sums up the elements it receives. Each time it receives an integer, it displays not only the integer itself, but also the sum it has computed up to then. From one execution of the handler to the other, we want to keep track of the already computed sum.

This is a weak kind of state. Actually, we can implement it using an accumulator. Let us first define a function working on a list that does the trick:

```
def displayAndSum(prevSum: Int)(list: List[Int]) {
  list match {
    case Nil => ()
    case head :: tail =>
      val sum = prevSum + head
      println(head + "\t-> " + sum)
      displayAndSum(sum)(tail)
  }
}
```

We can then pass this method as the handler for a port:

```
val port = Port.make(displayAndSum(0))
```

This is quite cumbersome. Actually, we can write it more elegantly using `foldLeft`:

```
def displayAndSum(init: Int)(list: List[Int]) {
  list.toAgent.foldLeft(init) { (prevSum: Int, element: Int) =>
    val sum = prevSum + element
    println(element + "\t-> " + sum)
    sum
  }
}
```

`foldLeft` is a standard method of collections. It starts from an initial value. It applies the given function on this value and the first element of the collection, giving a new value. It then repeats the operation with the second element, and so on, until the collection has been entirely traversed. It can be defined for lists as:

```
def foldLeft[B](init: B)(f: (B, A) => B) =
  if (this.isEmpty) init
  else this.tail.foldLeft(f(init, this.head))(f)
```

The reader not too familiar with functional programming techniques might find easier to understand this alternative, imperative implementation:

```
def foldLeft[B](init: B)(f: (B, A) => B) = {
  var accumulator = init
  for (element <- this)
    accumulator = f(accumulator, element)
  accumulator
}
```

The accumulator hidden in `foldLeft` makes up the state of the agent, while keeping the code of the agent declarative. This is a very nice property. Again, this is such a common use case of ports that we define another shortcut for it:

```
def newPortObject[A, B](init: B)(handler: (B, A) => B) =
  make[A](_.toAgent.foldLeft(init)(handler))
```

Using this builtin method, we finally rewrite our example as:

```
val port = Port.newPortObject(0) { (prevSum: Int, element: Int) =>
  val sum = prevSum + element
  println(element + "\t-> " + sum)
  sum
}
```

Chapter 4

Ozma programming techniques

The previous chapter has introduced Ozma and its concepts. This chapter provides examples of programming techniques in Ozma. Most examples in this chapter are translations of Oz examples in [VH04].

The examples are grouped into the following categories:

- Sequential list processing, taking advantage of the tail call compilation of list to ease the development in the functional subset of Scala,
- Deterministic concurrency, using streams, and
- Nondeterministic concurrency, using ports.

4.1 List processing and tail call compilation

4.1.1 Sorting a list with a merge sort

The first example we will look at is a merge sort algorithm. We assume familiarity with the algorithm in general. This example illustrates a non-trivial combination of list manipulations, implicit parameters, default values and careful tail recursion. The code is given in figure 4.1.

The program takes a list of integers on the command-line as input. It first convert the array of strings to a list of integers. It then uses the method `mergeSort` to sort the list, and finally prints the sorted elements on the standard output.

The `mergeSort` method accepts an implicit parameter specifying the ordering to apply to elements. In this case, in `main`, the typechecker knows that it works on a list of `Int`. Therefore, it will look for an object that conforms to `Ordering[_ >: Int]` in the implicit scope. The Scala standard library provides such an implicit, `Ordering.Int`. This ordering obviously follows the natural ordering of integers. The provided implicit is forwarded to `merge` and to the recursive call. More on implicits can be found in [Ode11, cpt. 7].

`mergeSort` uses two helper functions: `split` and `merge`. `split` takes a list as input and splits it in two lists. Its specification is that the concatenation of the two result lists is some permutation of the input list. `merge` merges two sorted lists into a new sorted list. Its specification is that the resulting list is sorted and contains all the elements of the two input lists.

`split` is implemented using two accumulators. It calls itself recursively for each element of the list. At each recursive call, it swaps the two accumulator lists, and appends the element to one of them. This is one of many ways to implement this method. Note that `split` has *default*

```
import scala.ozma._

object MergeSort {
  def main(args: Array[String]) {
    val list = args.toList map (_.toInt)
    val sorted = mergeSort(list)
    sorted foreach println
  }

  def mergeSort[A](list: List[A])(
      implicit cmp: Ordering[_ >: A]): List[A] = {
    list match {
      case Nil => Nil
      case head :: Nil => list
      case _ =>
        val (left, right) = split(list)
        merge(mergeSort(left), mergeSort(right))
    }
  }

  def split[A](list: List[A], leftAcc: List[A] = Nil,
      rightAcc: List[A] = Nil): (List[A], List[A]) = {
    if (list.isEmpty) (leftAcc, rightAcc)
    else split(list.tail, list.head :: rightAcc, leftAcc)
  }

  def merge[A](left: List[A], right: List[A])(
      implicit cmp: Ordering[_ >: A]): List[A] = {
    if (left.isEmpty) right
    else if (right.isEmpty) left
    else {
      if (cmp.lteq(left.head, right.head))
        left.head :: merge(left.tail, right)
      else
        right.head :: merge(left, right.tail)
    }
  }
}
```

Figure 4.1: Merge sort in Ozma

```
import scala.ozma._

object TokenRing {
  def main(args: Array[String]) {
    val count = args(0).toInt
    createTokenAgents(count)

    while (true) sleep(1000)
  }

  def createTokenAgents(count: Int) {
    def loop(id: Int, input: List[Unit]): List[Unit] = {
      if (id > count) input
      else loop(id+1, thread(tokenAgent(id, input)))
    }

    val bootstrap: List[Unit]
    val output = loop(1, bootstrap)
    bootstrap = () :: output
  }

  def tokenAgent(id: Any, inTokens: List[Unit]): List[Unit] = {
    if (inTokens.isEmpty) Nil
    else {
      println(id + " has the token")
      sleep(1000)
      inTokens.head :: tokenAgent(id, inTokens.tail)
    }
  }
}
```

Figure 4.2: Token Ring in Ozma

parameters for its two accumulators. If no actual parameter is provided for them when calling `split`, the default values will be used. This is a simple way of providing the initial value of the accumulators. Another possibility is to define a method with accumulators nested in the method without accumulators.

The `merge` method uses the ordering it receives as implicit parameter `cmp` to compare the heads of the lists. It builds its result using the tail-recursive property of the `::` operator.

4.2 Deterministic concurrency using streams

4.2.1 Token ring

As a first example of concurrent program, we present a simulation of a token ring. In a token ring, a set of active enties live in a circle, and there is one token. The entity that has the token can perform an operation. Then it gives the token to its neighbor in the ring, and waits until the token comes to it again, after having followed the entire ring.

The token is simply a value of type `Unit`, since its value serves no purpose. The entities are modeled by agents, communicating with streams. Since the only communication between the entities is passing the token around, the streams have `Unit` elements.

The code of the program is shown in figure 4.2. Since this program uses only immutable, dataflow values, it is deterministic.

In this simulation, each agents has an ID and runs the `tokenAgent` method. Each time it receives the token from `inTokens`, it displays its ID on the standard output. Then it waits for a second before outputting the token.

The method `createTokenAgents` is responsible for creating the agents, and establishing the ring. Creating the agents is trivial, but setting up the streams properly is worth an explanation.

Suppose there are 3 agents, A, B and C. We want that when A outputs the token, B receives it. Similarly, B passes the token to C, and C to A. The *loop* method takes an ID and an input stream as parameters. It creates an agent with this ID, binds its input to the specified `input`. It also gives its output as the input for the following agent. Recursion ends when the last agent has been created, at which point `loop` returns its output stream.

The tricky part is to bind the output of the last agent to the input of the first one. In order to do so, we use a single assignment value, `bootstrap`. Note that we do not bind `bootstrap` to `output` directly. Instead, we bind it to `() :: output`. This has two effects: we throw an initial token into the ring, and when the last agent will output its first token, it will arrive in *second* position in the input stream of the first agent.

Therefore, token streams are connected in a kind of *spiral*.

Note that waiting for the token to arrive at a given agent is implicitly done when calling `inTokens.isEmpty`, as this requires `inTokens` to be bound.

4.2.2 Bounded buffer

In section 3.3.3, we have used streams that are computed eagerly. In section 3.3.5, the streams were computed lazily. These two use cases are extremes, but we can compute streams in a semi-eager, semi-lazy fashion. This is the role of a *bounded buffer*.

A bounded buffer is a FIFO queue that has a maximal capacity. It has a producer and a consumer. The producer can get ahead of the consumer by inserting elements in the buffer. But if it fills the buffer, it has to wait until the consumer has made room in the buffer.

Because streams are mostly used in pipelines with filters or maps, we want a bounded buffer to be composed the same way, as a third kind of agent (besides filters and maps).

Figure 4.3 shows the code of a program using a bounded buffer. Once again, this program is totally deterministic.

The `main` method acts as the consumer. It creates the producer thread, which is a lazy function generating integers. It then creates a bounded buffer from the produced list, with a capacity of 5 elements. This immediately authorizes the producer to produce the 5 first integers.

The consumer then displays the first element. It needs to wait for half a second, because the producer needs that time to generate the first element. Since the first element has been consumed, this also allows the producer to produce a sixth element.

The consumer then waits 4 seconds (8 times half a second). Meanwhile, the producer continues to produce eagerly the integers 2 to 6 included. But it does not generate the number 7, because the buffer is full.

After the imposed delay, the consumer tries to display 10 additional elements. The first five items (2 to 6) can be displayed immediately, since the producer has already consumed them. But to display numbers 7 to 11, it needs to wait for the producer to produce them.

Now does the bounded buffer work? It is pretty easy, actually. The idea is to *look ahead* the input compared to the output. The first statement, line 27, spawns a thread that drops the `capacity` first elements of the list. In other words, it forces the first `capacity` elements to be needed, thereby triggering the producer for these.

Then, it loops with a tail-recursive function that outputs exactly the input in a lazy fashion. But each time one element is output, it also advances the *look-ahead* by one. Hence, `inputAhead` is

```
import scala.ozma._

object BoundedBuffer {
  val TimeUnit = 500

  def main(args: Array[String]) {
    val produced = generate(1)
    val buffered = thread(boundedBuffer(produced, 5))

    println(buffered(0))

    sleep(8*TimeUnit)
    for (i <- 1 to 10)
      println(buffered(i))
  }

  def generate(from: Int): List[Int] = byNeedFuture {
    sleep(TimeUnit)
    from :: generate(from+1)
  }

  def boundedBuffer[A](input: List[A], capacity: Int): List[A] = {
    def loop(input: List[A], inputAhead: List[A]): List[A] = byNeedFuture {
      input.head :: loop(input.tail, thread(inputAhead.tail))
    }

    val inputAhead = thread(input drop capacity)
    loop(input, inputAhead)
  }
}
```

Figure 4.3: Bounded Buffer in Ozma

always `capacity` items in advance compared to `input` (though it might not be already computed, i.e. be unbound).

Note that this implementation of the bounded buffer assumes that the input has infinite size.

4.2.3 Digital logic simulation

Let us look at a bigger example. We will simulate a digital logic circuit using streams. In a digital circuit, there are wires and gates. Each wire carries a discontinuous signal of bits (i.e., one bit at each instant of time). Each gate has one or more input signals and one output signal.

The Scala language is designed in such a way that it is possible to define internal Domain Specific Languages (DSL) inside the code. We would like a DSL for encoding digital circuits. The design of this DSL is treated in appedix B. Here we will simply write some tests with this DSL.

In this DSL, bits are representing by two case objects `Zero` and `One`, that extend the sealed class `Bit`. We do not use simple integers nor booleans because a) we want type safety and b) we want dedicated operations on bits.

Now, in digital logic, we do not work with simple bits, but on time-varying *signals*. In our simulation, we use *discrete* signals. Then a signal has a bit value at each discrete instant of time. Hence, we model a signal as a stream of bits, i.e. `List[Bit]`. For convenience and documentation purpose, we define a shortcut for this one:

```
type Signal = List[Bit]
```

Note that signals can be finite or infinite.

A full adder

A full adder is a small digital circuit that computes the sum of three binary digits, x, y and z. It outpus two bits, c and s, such that $x + y + z = (cs)_2$, where $(cs)_2$ denotes the number cs interpreted in binary.

One implementation of a full adder is shown in figure 4.4. We can translate it to the following formulae.

$$t = x \oplus y$$
$$c = (t \cdot z) + (x \cdot y)$$
$$s = t \oplus z$$

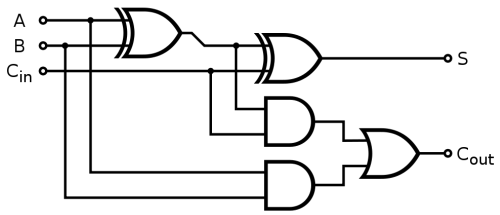


Figure 4.4: Digital circuit for a full adder

We can write these operations straightforwardly with the DSL:

```
val t = x ^^ y
val c = (t && z) || (x && y)
val s = t ^^ z
```

Note that because of shallow unification in Oz, these statements can be written in any order, while preserving a) the absence of deadlock and b) the neat result. For example, one can write:

```
val t: Signal
val c = (t && z) || (x && y)
val s = t ^^ z
t = x ^^ y
```

The former is however easier to write because we do not need to introduce a single assignment value.

We want to use a full adder as a separate abstraction. Thus we define it as a function:

```
def fullAdder(x: Signal, y: Signal, z: Signal) = {
  val t = x ^^ y
  val c = (t && z) || (x && y)
  val s = t ^^ z

  (c, s)
}
```

We can define the following test case in order to check that it works as expected:

```
val x = Signal(1, 1, 0)
val y = Signal(0, 1, 0)
val z = Signal(1, 1, 1)

val (c, s) = fullAdder(x, y, z)

assert(c == Signal(1, 1, 0))
assert(s == Signal(0, 1, 1))
```

A latch

A latch is a digital component that is able to store a bit of data. It has a data input and a control input. When the control is off, the input passes through the latch towards the output. When the control is on, the output keeps its previous value, thereby storing the last value that was in the input when the control was off.

The following function defines a latch:

```
def latch(control: Signal, input: Signal) = {
  val output: Signal
  val f = Gates.delay(output)
  val x = f && control
  val z = !control
  val y = z && input
  output = x || y
  output
}
```

It uses a *delay* gate. Each value of a delay gate is the value of its input at the previous time slot. Implementing `delay` is very easy:

```
def delay(input: Signal): Signal = Zero :: input
```

Note that in the latch, `f` is defined as a function of `output`, and `output` as a function of `f` (indirectly). We use a single assignment value to tie the loop. And it actually works fine because the gates that work on the streams are actually implemented as `agents`. The delay outputs immediately a 0. This 0 can be used as the first value of `f`, thereby allowing for the computation of the first value of `output`. This first value of `output` is reinjected as second value of `f` because of dataflow. So there is no dead lock.

The following test case ensures the latch behaves correctly:

```
val control = Signal(0, 0, 1, 1, 0, 0, 1)
val input   = Signal(1, 0, 0, 1, 1, 0, 1)

val output = latch(control, input)

assert(output == Signal(1, 0, 0, 0, 1, 0, 0))
```

Infinite signals

Usually, when simulating digital circuits, we want to model infinite signals. This can also be modeled with our DSL. The basic block is the *clock*: a component that issues 1's at a constant rate. The default period is 1 second.

Clocks can be used to control the rate at which a programmed generator yields values. The DSL provides a simple kind of generator that cycles through a sequence a values.

The following test uses a clock and cycles:

```
val clock = Signal.clock()
val left = Signal.cycle(clock, 1, 0, 1)
val right = Signal.cycle(clock, 0, 1)
val output = left && right
```

Note that we cannot check that the result is the one expected with an `assert`, because the signals are infinite. Instead, we use a `display` method that is able to display a set of time-varying signals on the standard output (see figure 4.5). We use it like this:

```
display('l' -> left, 'r' -> right, 'o' -> output)
```

The test displays the following:

```
l r o

1 0 0
0 1 0
1 0 0
1 1 1
0 0 0
1 1 1
1 0 0
0 1 0
1 0 0
1 1 1
...
```

The full examples, as well as a digital clock displaying in binary form the number of seconds since program start are available in the directory `docs/examples/digitallogic/`.

4.3 Nondeterministic concurrency using ports

4.3.1 Port objects: the Tossing the Ball game

In the Tossing the Ball game, the players toss each other a ball. Each time a player receives a ball, it chooses one of the other players, and tosses him the ball. Figure 4.6 show the code of a simple program simulating this game with port objects.

```
import scala.ozma._
import ozma._

import digitallogic._

object Utils {
  /**
   * Display a set of digital signal
   * Each given signal is a column on the standard output.
   * It begins with the header and then its values at each
   * point of time.
   */
  def display(signals: (Char, Signal)*) {
    def loop(signals: List[Signal]) {
      val next = for (signal <- signals) yield {
        if (signal.isEmpty)
          return

        print(signal.head + " ")
        signal.tail
      }

      println()
      loop(next)
    }

    val sigs = signals toList

    for (header <- sigs.map(_._1))
      print(header + " ")

    println()
    println()

    loop(sigs.map(_._2))
  }
}
```

Figure 4.5: Display a set of signals on the standard output

```
import scala.ozma._
import ozma._

object TossingTheBall {
  val TimeUnit = 1000

  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(TimeUnit)
  }

  def makePlayer(id: Any, others: Seq[Player]): Player = {
    Port.newStatelessPortObject { ball =>
      println(id + " received the ball")
      sleep(TimeUnit)
      Random.rand(others).send(ball)
    }
  }
}
```

Figure 4.6: Tossing the Ball game in Ozma

We define a ball, of type `Ball` as being a simple token, hence it is equivalent to the `Unit` type. A player is a port that can be sent balls, hence the `Player` type is equivalent to `Port[Ball]`.

The `makePlayer` creates a player, given an ID an the other players it is supposed to play with. Using the `newStatelessPortObject` abstraction, it is easy to define its behavior. Each time it receives a ball, it displays its ID, waits for a second, then send the ball to one of the other players, at random.

The `main` method uses three single assignment values to set up the relations between the players. It then sends the ball to the first player. Running this program will display something like this:

```
Player 1 received the ball
Player 3 received the ball
Player 1 received the ball
Player 3 received the ball
Player 2 received the ball
Player 1 received the ball
Player 3 received the ball
...
```

Notice how the usage a port allows a player to receive the ball from a set of other players, in a nondeterministic way.

4.3.2 Functional building blocks as concurrency patterns

Ports can be used to implement many message protocols. The previous example showed a simple protocol with a single kind of message. And yet the handler code of a port object is a declarative function working on a list.

Actually, because the message-passing model is so close to the declarative model, we can implement complex message-passing protocols using simple declarative methods. The collection library provides many methods that can be used like this. In particular, we can use method of `List` to implement complex message-passing protocols.

Consider this *server port*: basically a port that is able to *answer* to the messages it receives. It is a computation server, taking care of some computation on behalf of its client.

```
import scala.ozma._
import ozma._

object ServerPort {
  val computerPort = ResultPort.newStatelessPortObject {
    x: Int => 3*x*x + 2*x - 4
  }

  def main(args: Array[String]) {
    val result = computerPort.send(3)
    println(result) // displays 29
  }
}
```

We can make this server a little more generic, allowing to parametize it with three coefficients:

```
import scala.ozma._
import ozma._

object ServerPort {
  def makeSecondDegreeServer(a: Int, b: Int, c: Int) = {
    ResultPort.newStatelessPortObject {
```

```
import scala.ozma._
import ozma._

object ServerPort {
  def makeSecondDegreeServer(a: Int, b: Int, c: Int) = {
    ResultPort.newStatelessPortObject {
      x: Int => a*x*x + b*x + c
    }
  }

  def main(args: Array[String]) {
    val ports = List(
        makeSecondDegreeServer(3, 2, -4),
        makeSecondDegreeServer(1, 2, 3),
        makeSecondDegreeServer(5, -4, -4),
        makeSecondDegreeServer(0, 0, 3),
        makeSecondDegreeServer(3, 2, 1),
        makeSecondDegreeServer(-1, -1, -1)
    )

    val results = ports map (_ send 3)
    results foreach println

    println("max: " + results.max)
  }
}
```

Figure 4.7: Server ports in Ozma

```
      x: Int => a*x*x + b*x + c
    }
  }

  def main(args: Array[String]) {
    val port = makeSecondDegreeServer(3, 2, -4)
    val result = port.send(3)
    println(result) // displays 29
  }
}
```

The last step is to create a bunch of these server ports, with different parameters.

We then want to broadcast a message to all of them, and collect the results. This can be done very easily with a simple method of `List`, i.e. `map`. Figure 4.7 shows the resulting code.

This is but a rudimentary example. Collection methods can provide very easy implementation of complex message-passing protocols. We can for example use `max` on the resulting list to keep the highest result, as shows the last line of the code.

4.3.3 The Flavius Josephus problem

The Flavius Josephus problem is a well-known problem of computer science and mathematics. The problem states as follows. There are N persons standing in a circle. Starting from person number 1, we count K persons. The Kth persons loses the game (in the original version, dies), and we start counting again at the person following her. Hence, every Kth person is removed from the circle. The problem is to find the number of the person who will stay last in the circle.

We can solve this problem computationally quite easily. We create a class `Victim` for each person in the circle. Each `Victim` has a reference to the next person in the circle and the previous

```
def josephus(count: Int, step: Int) = {
  class Victim(val id: Int) {
    private var alive: Boolean = true
    private var succ: Victim = _
    private var _pred: Victim = _

    def pred = _pred
    def pred_=(value: Victim) {
      _pred = value
      _pred.succ = this
    }

    def kill(i: Int, survivors: Int): Int = {
      if (alive) {
        if (survivors == 1) {
          id
        } else if (i % step == 0) {
          alive = false
          succ.pred = pred
          succ.kill(1, survivors-1)
        } else {
          succ.kill(i+1, survivors)
        }
      } else
        succ.kill(i, survivors)
    }
  }

  val victims = for (id <- (1 to count).toList)
    yield ResultPort.newActiveObject(new Victim(id))

  val first = victims.head
  val last = victims.tail.foldLeft(first) { (pred, victim) =>
    victim.pred = pred
    victim
  }
  first.pred = last

  victims.head.kill(1, count)
}
```

Figure 4.8: The Flavius Josephus problem, active object version

person. We model a victim as an *active object*. An active object is a special view of an object. Each call to a method will be converted as a message send to a *port*. Hence, an active object implicitly defines a stateful port object, with a typed interface.

Figure 4.8 shows the code of this problem. It starts by sending a message `kill(1, count)` to the first victim. This victim forwards the message `kill(2, count)` to her successor, and so on, until we arrive at `kill(step, count)`, at which point the victim is killed. The victim removes herself from the circle by updating the predecessor of her successor (which is *also* a message send).

Figure 4.9 shows an alternative implementation of this problem that uses only the *declarative* subset of Ozma. It uses streams for communication between the agents, instead of port. This is possible because every agent has only one predecessor who can send it messages. Sometimes the predecessor changes, but there is always only one.

This shows that Ozma extends both the functional and object-oriented paradigms of Scala to easy concurrency.

```
def declarativeJosephus(count: Int, step: Int) = {
  type KillMsg = (Int, Int)

  val last: Int

  def victim(id: Int, stream: List[KillMsg]): List[KillMsg] = {
    if (stream.isEmpty) Nil
    else {
      val (i, survivors) = stream.head

      if (survivors == 1) {
        last = id
        Nil
      } else if (i % step == 0)
        (1, survivors-1) :: stream.tail
      else
        (i+1, survivors) :: victim(id, stream.tail)
    }
  }

  val initKillMsg = (1, count)

  val lastStream: List[KillMsg]
  lastStream = pipe(initKillMsg :: lastStream, 1, count)(victim)

  last
}

private def pipe[A](stream: List[A], id: Int, count: Int)(
    handler: (Int, List[A]) => List[A]): List[A] = {
  if (id > count) stream
  else pipe(thread(handler(id, stream)), id+1, count)(handler)
}
```

Figure 4.9: The Flavius Josephus problem, declarative version

4.3.4 Capture the Flag: an end-of-term Oz project

Our final example is much larger. It is a simulation of the game Capture the Flag. This was the end-of-term project of the second Oz course given at the Université Catholique de Louvain by Peter Van Roy in 2008–2009. This course is given to third-year students in computer science.

The project is focused on message-passing concurrency, with ports. This is the *only allowed* form of state in the entire code. We have taken our implementation in Oz, from 2008, and have translated it into Ozma. Of course, we have respected the requirements: there is not a single variable in the entire program.

The full source code can be found at https://github.com/sjrd/capture-the-flag, or on the CD-ROM accompanying the printed version of this master thesis. It includes scripts to easily compile and run it:

```
$ chmod +x compile run
$ ./compile
$ ./run
```

We will not discuss the implementation of this program. This is out of the scope of this text. The purpose of this translation was to show, by example, that Ozma is expressive enough to meet the requirements of the Oz project.

Chapter 5

Semantics of Ozma

The previous chapter has presented the features available in Ozma as a tutorial. Now we define more precisely the semantics of these features. Most of it is ruled by dataflow values and their status. Other features are built on top of that.

The semantics of Ozma are defined as a delta with respect to Scala semantics. These are defined in a quite rigorous, although informal way in [Ode11]. Note that Ozma is designed as a superset of Scala, so that every Scala program behaves the same if compiled with Ozma (modulo implementation restrictions).

The reader familiar with Oz should keep in mind the following naming differences:

Ozma	Oz
Value	Variable
Variable	Cell
Actual value	Value

Scala and Oz have different names for similar concepts. We chose to keep the vocabulary of Scala, because Ozma is closer to Scala than to Oz.

5.1 Dataflow values and suspension

The most important semantic change introduced in Ozma are dataflow values. In Ozma, every `val` is a dataflow value, and every `var` is a mutable placeholder that references a dataflow value. Dataflow values are defined by their status.

5.1.1 Value status

At any given time in the program, a dataflow value must have exactly one of the following statuses:[1]

- Unbound (initial state of all values),
- Unbound and needed,
- Determined and needed or

[1] In Oz there is a fifth status, kinded, that is not used in Ozma.

- 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. Once a value has been needed, it will never be non-needed again. Once it has been determined, it will never get unbound of failed again. The possible status transitions of a value are depicted in figure 5.1.

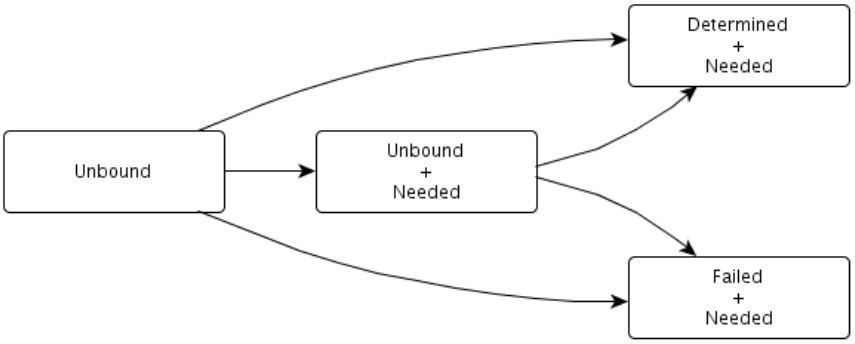


Figure 5.1: Possible status transitions for dataflow values

When a value is determined, it also holds the *actual value* it has been bound to. We use the term actual value to name what most languages, Oz included, simply name a value. 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 also holds a reference value to a throwable (an instance of `Throwable` of one of its subclasses). Note that *any* value can become a failed value, even if its static type[2] is not a supertype of `Throwable`. In fact, although the throwable itself has type `Throwable`, the wrapping has type `Nothing`. Now `Nothing` is a subtype of all types in Scala (in typing theory this corresponds to the *bottom* type). Hence this is well-typed.

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).

5.1.2 Primitive operations on value status

Initially, a value is unbound, and the sets of threads waiting for it are empty. The following primitive operations are defined on a value x. All operations on a value are atomic (cannot be interrupted by another thread), unless 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: raise the throwable wrapped inside x.

[2]The static type may be either stated explicitly or inferred by the typechecker.

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.

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 *be the same value*, with the more specific (the farther to the right) of the statuses of x and y, 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, resume all threads in the wait-for-needed set and empty it. If the resulting value is bound, resume all threads in the wait-for-bound set and empty it.
- If both x and y are failed: raise 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, raise a failure exception.

5.1.3 Building bound values

With the previous rules, there is no other way to make a bound value than from another bound value. So how do we initially build bound values? There are three ways of building determined values: literal constants, arithmetic and logic operations, and the `new` operator. They all create a new value that is determined, and that holds respectively the constant, the result of the operation, and a reference to the new instance.

There is also a primitive operation, `makeFailedValue(th: Throwable)` that builds a failed value wrapping the given throwable.

5.1.4 Implicit waiting

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` normally waits for its operands, but can continue without blocking if they have previously been unified (directly or indirectly).
- Other comparisons, boolean 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.

5.1.5 Status of boxed values

Recall from section 1.3 that 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. This behavior is well defined when the value to box/unbox is determined. But what happens if we box/unbox an unbound or failed value?

We define the behavior of boxed and unboxed values 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/unboxed value (as defined in the Scala specifications).
2. If x is failed, then y is unified to x, i.e. 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.

Note that in the last case, we do not define what happens if y becomes bound. This does not make sense, since y is the *result* of the boxing/unboxing operation, and should not be bound from the outside. The typechecker will ensure that this situation cannot happen, except in some obscure corner cases.

If it were to happen, the behavior can be deduced from the stated rules. Indeed, recall that any bound value is also needed. So rule 3b applies and x eventually becomes needed. This will resume any thread in the wait-for-needed set of x. Besides, if at any time x becomes bound after that, then rule 3c will make a new boxed/unboxed value and bind it to y according to rules 1 and 2.

For the unbox case, this will raise a failure iff the actual primitive values are different. For the box case, this will always fail if the primitive values are different; but it can either succeed or fail when they are the same, depending on the caching strategy of boxed instances.[3]

We might consider to define `=` so that it follows boxes naturally. This would solve this problem, and simplify boxing and unboxing semantics. However, given the present implementation of Oz, this definition would be impractical to implement. A future work could modify the behavior of the Mozart runtime engine to make this behavior straightforward to implement.

Example

Consider the following example:

```
import scala.ozma._

object Test{
  def compute(value: Int): Option[Int] = {
```

[3]This caching strategy is left undefined in the present document.

```
5       val y = byNeed {
          println("computing y")
          value * value // make value (hence, x) needed here
        }
        Some(y) // boxing here
10    }

      def main(args: Array[String]) {
        val x = byNeed {
          println("computing x")
15        5
        }

        val opt = compute(x)

20      println("start")
        if (opt.isEmpty)
          println("<none>")
        else {
          val z = opt.get // unboxing here
25        println(z) // rebox, internally making the box needed
        }
      }
    }
```

This example illustrates the most common, yet complex, impact of our definition. It starts at line 13, defining `x` as being a lazy value that evaluates to 5. On line 18, it stores in `opt` the result of the function `compute`, which is passed `x` as parameter.

This function declares `y` at line 5 as being a lazy value evaluating to `value * value`. Since `value` is `x`, this means `x*x`. It then wraps `y` in an instance of `Some`. This implies a boxing operation (boxing 1), since it enters a value of generic type. Because `y` is unbound, rule 3 applies, and this gives an unbound value that we call `y'`.

Back in the `main` method, we test `opt` for emptiness. Obviously it is not empty, so we get to line 24, where we unbox `y'` in `z` (unboxing 2). Since `y'` is unbound, rule 3 applies again and `z` is a new unbound value (that is not related to `y`, by the way).

On line 25, we display `z`. Since `println` expects a parameter of type `Any`, `z` has to be boxed again, yielding `z'` (boxing 3).

Now, in `println`, we call the method `toString` of `z'`, thereby making `z'` *needed*. This triggers rule 3b of boxing 3. So eventually, `z` will be made needed. By the same rule of unboxing 2 and then boxing 3, this makes `y'` then `y` needed.

This triggers the evaluation of `y`, thus evaluating `value * value`, i.e. `x*x`. Since `x` is needed, its evaluation is triggered, yielding 5. Hence, `y` becomes bound to 25.

This triggers rule 3c of boxing 1, and eventually rule 1 for the same boxing, thereby binding `y'` to a boxed 25. By successive application of the same rules on unboxing 2 and boxing 3, this binds a primitive 25 to `z` and then another boxed 25 to `z'`.

Finally, the `toString` method can execute, returning the string `"25"`, that can be displayed.

This example illustrates why the complex definition of boxing and unboxing operations is needed in the presence of unbound values and by-need execution. It also highlights that these rules provide transparency of boxing and unboxing operations, as is expected.

Fooling the typechecker

We illustrate how to fool the typechecker with the following code snippet:

```
val x = byNeed(5)
```

```
val y: Any
y = x // y becomes a unbound boxed version of x
y = 5 // we bind y from the outside to a boxed 5
```

We use a single-assignment value to break the specifications, exploiting the bidirectionality of value binding. `y` is bound twice to what should be the number 5. So in theory this is OK. But because of the boxing of the values, they are actually different.

5.2 Variables

Variables are introduced by the `var` keyword instead of `val`. A variable is a stateful placeholder that contains a dataflow value. Initially, it contains an unbound value.

The statement `x = y` where x is a variable is called an `assignment`. It is not a unification. Its effect is to throw out the current dataflow value contained in x, and replace it by y.

Any other use of a variable is implicitly dereferenced as the dataflow value it contains at the time the evaluation executes. This never blocks.

5.3 The `thread` primitive

The `thread` primitive is used to create a new thread. The expression `thread(body)` where `body` has type T is defined as follows.

If T is the `Unit` type, then this expression is a statement equivalent to the following Scala code:

```
new Thread(new Runnable {
  def run() = body
}).start()
```

If T is any other type, then this expression is equivalent to the following block:

```
{
  val x: T
  thread {
    x = body
  }
  x
}
```

5.4 `byNeed` and `byNeedFuture`

`byNeed` and `byNeedFuture` are not primitives. We have already implemented them with the help of the other primitives in section 3.3.5. We repeat them here for convenience.

```
def byNeed[A](value: => A) = {
  val result: A
  thread {
    waitNeeded(result)
    result = value
  }
  result
}
```

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

5.5 Ports

Ports are made 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 (private val rawPort: Any) {
  @native def send(element: A): Unit = sys.error("stub")
}

object Port {
  @native def newPort[A]: (List[A], Port[A]) = sys.error("stub")
}
```

Calling the `send` method of a port,

```
port.send(element)
```

has the following effects:

- 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`.

These three operations behave as if they were atomic.

Concretely, this means that `port.send(element)` appends `element` at the end of the stream bounded to `port`.

Other port creation methods are defined in terms of the `newPort` primitive:

```
object Port {
  // [...]

  def make[A](handler: List[A] => Unit) = {
    val (stream, port) = newPort[A]
    thread {
      handler(stream)
```

```
    }
    port
  }

  def newStatelessPortObject[A, U](handler: A => U) =
    make[A](_.toAgent foreach handler)

  def newPortObject[A, B](init: B)(handler: (B, A) => B) =
    make[A](_.toAgent.foldLeft(init)(handler))
}
```

5.6 Tail call optimization

This last section covers an optimization that the compiler is *required* to implement. Indeed, the application of this optimization affects the semantics of the code.[4] This is the tail call optimization on method application.

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

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

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

Somewhere in the program (maybe 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, ..., aj, ..., ai, ..., an)
  }
}
```

Let B be the set of indices corresponding to actual parameters that are themselves a call to a method[5]. An expression is a method call if and only if it is not one of (a) a constant literal or (b) a local value or variable or (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 actually 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.

Then 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 *highest* (right-most) element of C, i.e.[6]

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

[4] Well, a purist would then not call that an optimization ...

[5] This includes accessor methods!

[6] This strategy is arbitrary: any other priority convention would have been applicable. We chose right-to-left priority because *usually*, right-most parameters are more subject to recursion than others. A trivial example is the `::` class: recursion always happens on the tail of the list, which is the right-most parameter.

`ai` is of the form `meth(params...)`.

Then the call to m is replaced by a block almost equivalent to

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

That is to say, the method call in the argument is moved *after* the call to m (here is the effect on semantics). It is not exactly equivalent to the above code, however, because it is additionaly made tail-recursive with respect to `meth`.

5.6.1 The `@tailcall` annotation

We use the `@tailcall` annotation at definition site of methods to mark parameters of a method as being *unbound-safe*. Concretely, in the method

```
def someMethod(x: Int, y: Int) = (x, y+1)
```

x is unbound-safe because the execution of `someMethod` does not wait on x. However, y is needed for the addition, and is thus not unbound-safe.

We can expose this fact to callers of `someMethod` by annotating x with `@tailcall`:

```
def someMethod(@tailcall x: Int, y: Int) = (x, y+1)
```

This will allow the compiler to tail call optimize calls to `someMethod` with respect to the first parameter, but not the second one.

5.6.2 `@tailcall` and case classes

Case classes receive particular attention from the compiler with respect to the `@tailcall` annotation. When defining a case class *with no constructor code*, all the parameters of the constructor are automatically `@tailcall`-annotated by the compiler. Consider the code of figure 5.2, which is taken from the file `docs/examples/trees/BinaryTrees.scala`.

The case class `Node` does not have any constructor code. Hence, the three constructor parameters `value`, `left` and `right` are automatically `@tailcall`-annotated. This makes the two recursive calls in `insert` to be tail call optimized, without any special indication in the source code.

In the standard library, the following case classes have `@tailcall` annotations:

- Class `::` (subclass of `List`),
- Class `Some` (subclass of `Option`),
- Classes `Left` and `Right` (subclasses of `Either`),
- All tuple classes.

Constructor code

The following case class definitions do not have constructor code, and hence get the `@tailcall` annotation:

```
abstract class Tree[+A]
case object Leaf extends Tree[Nothing]
case class Node[+A](value: A, left: Tree[A] = Leaf,
    right: Tree[A] = Leaf) extends Tree[A]

object BinaryTrees {
  type Comparer[-A] = (A, A) => Boolean

  def insert[A](smaller: Comparer[A])(tree: Tree[A],
      value: A): Tree[A] = tree match {
    case Leaf => Node(value)

    case Node(v, left, right) =>
      if (smaller(value, v))
        Node(v, insert(smaller)(left, value), right) // this is tail-recursive
      else
        Node(v, left, insert(smaller)(right, value)) // this is tail-recursive
  }

  def displayTree(tree: Tree[Any]) {
    tree match {
      case Leaf => ()
      case Node(value, left, right) =>
        displayTree(left)
        println(value)
        displayTree(right)
    }
  }

  def insertMany[A](smaller: Comparer[A])(tree: Tree[A],
      values: TraversableOnce[A]) =
    values.foldLeft(tree)(insert(smaller))

  def main(args: Array[String]) {
    val tree = insertMany((_:Int) < (_:Int))(Leaf, List(5, 1, 9, 8, 9, 6, 2, 10))
    displayTree(tree)
  }
}
```

Figure 5.2: `@tailcall` and case classes

```
class SomeSuperClass(x: Int)

// trivial case class with no body at all
case class Good1(x: Int)

// parameters to the super constructor are identifiers
case class Good2(x: Int) extends SomeSuperClass(x)

// var fields with no initial values
case class Good3 {
  var field: Int = _
}

// methods and inner classes are ok
case class Good4(x: Int) {
  def someMethod = x + 1

  class Inner {
  }
}
```

The following case classes definitions would not get `@tailcall` annotations because they have constructor code (directly or indirectly):

```
case class Bad1(x: Int) {
  println(x) // statement in the constructor
}

// a super constructor parameter is not an identifier
case class Bad2(x: Int) extends SomeSuperClass(x+1)

// value or variable field with initialization
case class Bad3(x: Int) {
  val someField = x
}

// inner object
case class Bad4(x: Int) {
  object innerObj {
  }
}
```

5.6.3 Example: tail-recursion modulo cons

Since the case class `scala.collection.immutable.::` has no constructor code, all its constructor parameters are `@tailcall`-annotated. Moreover, the compiler is required to inline the `::` method of the `List` type, in order to support tail-recursion on lists.

Consider the definition of `append()` from section 3.3.2.

```
def append[A](front: List[A], back: List[A]): List[A] = front match {
  case Nil => back
  case head :: tail => head :: append(tail, back)
}
```

Inlining the `::` operator, the last line becomes

```
  case head :: tail => ::(head, append(tail, back))
```

Here the three conditions for `@tailcall` optimization meet:

- The call to `::` is in tail position in the body of `append()`,

- The second actual parameter is a method call, and
- The second formal parameter of `::` is `@tailcall`-annotated.

Hence, the compiler first rewrites the call to have an explicit result, as output parameter (this is no valid Ozma code, it is internal only):

```
def append[A](front: List[A], back: List[A], out result: List[A]) {
  front match {
    case Nil => result = back
    case head :: tail =>
      result = ::(head, append(tail, back))
  }
}
```

Finally, the last statement is rewritten using a single-assignment value, so that the call to `append()` is done after the call to `::`.

```
def append[A](front: List[A], back: List[A], out result: List[A]) {
  front match {
    case Nil => result = back
    case head :: tail =>
      val resultTail: List[A]
      result = ::(head, resultTail)
      append(tail, back, resultTail)
  }
}
```

5.6.4 Discussion

We finish this section on `@tailcall` optimization with a discussion on the rationale that lead to these semantics.

In Oz, there is no `@tailcall` annotation: the Oz compiler applies that kind of tail call optimization on any *record construction*, which includes building a cons pair, without requiring the programmer to state that some parameters are safe with the annotation.

The Oz compiler can do that because record construction is a language construct that has fixed semantics. And this semantics never waits on the fields of the record. Hence, it is always safe to tail call optimize record construction. In the light of the previous discussion, you might consider that the Oz compiler automatically annotates all parameters of record construction statements.

In Ozma, data structures are classes, which have user-defined constructors, and hence *no fixed semantics*. The compiler, by itself, does not know whether a particular parameter of a constructor is ready to accept unbound values. This is why we require the programmer to annotate manually parameters that are known to be safe.

One might nevertheless ask the following question: could the compiler *infer* that a given parameter is unbound-safe? This would be considerably easier for the programmer. The answer to this question is no, for two reasons.

First, inferring this is an undecidable problem. The proof is straightforward from the fact that it is a property of the semantics of the code. We could write an approximate algorithm that adds the annotation when it can decide that it is safe, and does not when it cannot tell (safe approximation). But that would be impossible to define precisely, and hence one could not rely on the fact that a particular parameter will be `@tailcall`-annotated.

A second reason is simply overriding. Consider a method m in class `ParentClass` with a parameter p that is inferred to be safe. Now consider a class `ChildClass` extending `ParentClass`, that overrides m in a way that makes p *unsafe*. Then we cannot tail call optimize m on instances of `ChildClass`, though we can for `ParentClass`. This breaks the Liskov substitution principle [LW94].

This is why we need `@tailcall` to be applied manually.

However, as a section 5.6.2 explained, the compiler *does* annotate automatically some parameters, namely the parameters of case class constructors that have no code. This applies in clearly specified situations that imply unbound-safety. This is both sound and predictable. Moreover, the overriding argument does not apply to constructors, since constructors are never inherited, nor, a fortiori, overridden.

When using case classes as a corresponding feature to Oz records, these case classes are tipically trivial (one-line), and do not have constructor code. Hence, they get `@tailcall`-annotated.

The resulting system is thus as convenient as tail call optimization in Oz. And in addition, it can be more powerful if needed, allowing a programmer to specify any user-defined method to have `@tailcall`-annotated parameters.

Chapter 6

Implementation of Ozma

The previous chapter has first given an overview of the features of Ozma. Then it has precisely defined their semantics. The present chapter will present the key aspects of the implementation of Ozma.

As are Scala and Oz, Ozma is a semi-compiled, semi-interpreted language. Its compiler is written in Scala, while its interpreter is written in Oz. The interpreter is rather straightforward and can be found in `src/engine/`. Its only role is to load the requested module and call its `main` method. All the rest is taken care of by the interpreter of Mozart and the runtime library.

We will therefore not discuss the interpreter in this document, but will rather concentrate on the compiler, which does all the work.

The Ozma compiler is based on the official Scala compiler. Actually, it is a pure *extension*, in the OO sense, of classes of the Scala compiler. This chapter will first present the relevant aspects of the Scala compiler. Then it will move on to the details of compiling Ozma.

6.1 The Scala compiler

The current Scala compiler is `nsc`, which stands for New Scala Compiler. It is written itself in Scala. Basically, it consists of a sequence of *phases*. There is also a global symbol table that is extensively described in [Ode09].

The architecture of `nsc` is very complex, making extensive use of mixin composition, path-dependant types, and other advanced features of Scala itself. An introduction to this architecture can be found in [OZ05, sec. 4].

6.1.1 Phases

The first phase is the parser: it reads the source file and builds the AST. Subsequent phases transform the AST, simplifying it at each step, until the resulting AST is simple enough to make its translation to *icode* straightforward. Icode is a platform-independent language for stack-based virtual machines. Additional phases further transform the icode, for example for optimization purposes. The last phase is responsible for converting the icode into the platform-dependant bytecode (the one of the JVM or MSIL).

In a nutshell, we have:

1. The `parser` phase: build the original AST,
2. The `namer` phase: assign names to all identifiers,

3. The `typer` phase: give a type to every node of the AST,
4. Several transformation phases: sucessively simplify the AST,
5. The `icode` phase: generate icode from the AST,
6. Several optimization phases: successively transform the icode,
7. The `jvm` phase: produce actual JVM bytecode.

The complete list can be obtained by running:

```
$ scalac -Xshow-phases
```

All phases between the `typer` and `icode` phases work with type-annotated ASTs. They do most of the translation work.

What is wonderful about `nsc` is that we can add, replace, or even delete phases as we want. This is how the Ozma compiler is built. It removes all phases from `icode` to `jvm`, and replaces them with phases that convert the AST towards Oz ASTs. Additionaly, it removes or replaces transformation phases that are not appropriate (such as the original `tailcalls` phase), and adds some specific phases (such as `looprecover`).

Appendix A describes the role of each phase in the Scala/Ozma compiler.

6.1.2 A simple phase: while loop recovery

In order to get some insight about what a phase looks like, we will first describe the simplest of all phases: `looprecover`. This phase is required by the Ozma compiler in order to recover `while` and `do..while` loops. Indeed, the parser converts them into `if`'s and **goto**'s. Since there is no goto in Oz, we must get rid of these.

The `looprecover` phase scans the AST for occurrences of the subtree pattern that correspond to loops. It replaces them by calls to the runtime methods `whileLoop` and `doWhileLoop` defined in the package object `scala.ozma`.

Consider the following code snippet (`docs/examples/echo/Echo.scala`):

```
object Echo {
  def main(args: Array[String]) {
    var i = 0
    while (i < args.length) {
      Console.println(args(i))
      i += 1
    }
  }
}
```

The parser outputs an AST that is equivalent to the following internal source code for the `main` method:

```
def main(args: Array[String]): scala.Unit = {
  var i = 0;
  while$1(){
    if (i.$less(args.length))
      {
        {
          Console.println(args(i));
          i.$plus$eq(1)
```

```
        };
        while$1()
      }
    else
      ()
  }
}
```

The `looprecover` phase rewrites this AST so that it calls `whileLoop` instead of defining labels and jumps, giving:

```
def main(args: Array[String]): scala.Unit = {
  var i = 0;
  _root_.scala.ozma.whileLoop(i.$less(args.length))({
    Console.println(args(i));
    i.$plus$eq(1)
  })
}
```

The `whileLoop` itself is defined in the standard library of Ozma as follows:

```
def whileLoop(cond: => Boolean)(body: => Unit) {
  if (cond) {
    body
    whileLoop(cond)(body)
  }
}
```

Figure 6.1 shows the entire code of the `looprecover` phase. Each phase is actually encapsulated in a *component* (subclass of `nsc.SubComponent`), that is a factory for the actual phase. Line 11 introduces the component class for the `looprecover` phase. It inherits from class `Transform`, which is a subclass of `SubComponent` specialized for phases that do some kind of *transformation* of the AST. This is exactly what we need here.

The `newTransformer` method (line 16) is the factory method that creates the actual phase. The phase class is `WhileLoopRecoverer`, beginning at line 22. This one extends `Transformer`, which is a subclass of `Phase` that has a role similar to `Transform`.

The actual job is done by the `transform` method (line 32). It takes a subtree of the AST (of class `Tree`), and returns the (possibly) transformed subtree. The superclass `Transformer` defines `transform` so that it calls itself recursively on all sub-subtrees of the subtree. This gives a very easy framework to transform parts of an AST.

So what does `transform` do? The usual form of a `transform` is to match the `tree` parameter against AST patterns that it needs to transform. The `looprecover` phase recognizes while-like label definitions and their respective goto's. Figure 6.2 shows the subtree before its transformation, and after it has been converted to a call to `whileLoop`.

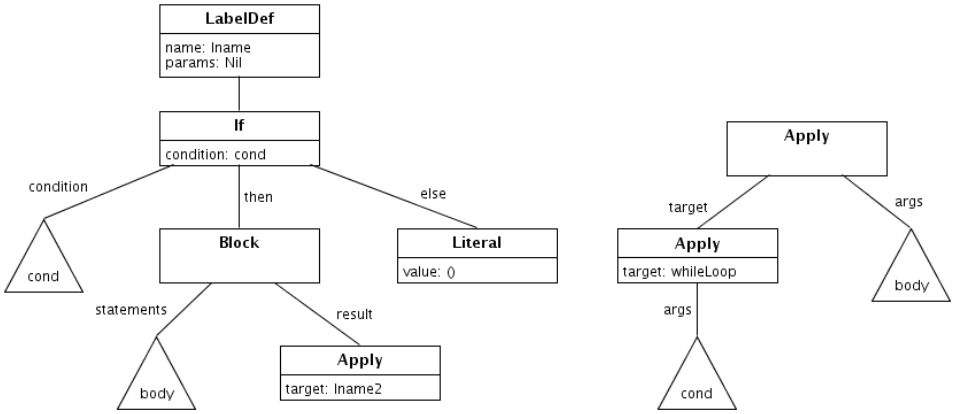
One might think that identifying the subtrees that "look like" the tree of figure 6.2(a) would require a lot of tedious code. But here the advanced pattern matching feature of Scala come in handy. A single `match` statement is able to match everything, and extract at the same time the two variable subtrees: `cond` and `body`. That is what is done at line 35.

The body of this `case` statement is only a matter of reconstructing the subtree of figure 6.2(b), while keeping the positions from the source code. The methods of `treeCopy` create new AST nodes from an existing one (the first argument), copying its source position and symbol information. Here we create two nested `Apply` nodes (method calls).

A similar transformation applies for `do`..`while` loops.

```
package scala.tools.nsc
package ozma

import scala.collection.{ mutable, immutable }
import transform.Transform

/** This component recovers while and do..while loops that the parser destroys
 *  in labels and jumps. It replaces them by calls to `scala.ozma.whileLoop'
 *  and `scala.ozma.doWhileLoop', respectively.
 */
abstract class WhileLoopRecovering extends Transform {
  import global._

  val phaseName: String = "looprecover"

  def newTransformer(unit: CompilationUnit): Transformer =
    new WhileLoopRecoverer(unit)

  /**
   * Recover while loops that the parser has destroyed into labels and jumps
   */
  class WhileLoopRecoverer(unit: CompilationUnit) extends Transformer {
    import symtab.Flags._

    def scalaOzmaDot(name: Name) =
      Select(gen.rootScalaDot("ozma"), name)

    /** Rewrite while-like label defs/calls as calls to
     *  `scala.ozma.whileLoop' and do-while-like label defs/calls as calls to
     *  `scala.ozma.doWhileLoop'.
     */
    override def transform(tree: Tree): Tree = {
      tree match {
        // while (cond) { body }
        case LabelDef(lname, Nil,
            If(cond,
                Block(List(body), Apply(Ident(lname2), Nil)),
                Literal(_))) if (lname == lname2) =>
          val whileLoop = atPos(tree.pos)(scalaOzmaDot("whileLoop"))
          val innerApply = treeCopy.Apply(tree, whileLoop, List(cond))
          val outerApply = treeCopy.Apply(tree, innerApply, List(body))
          outerApply

        // do {body} while (cond)
        case LabelDef(lname, Nil,
            Block(List(body),
                If(cond,
                    Apply(Ident(lname2), Nil),
                    Literal(_)))) if (lname == lname2) =>
          val doWhileLoop = atPos(tree.pos)(scalaOzmaDot("doWhileLoop"))
          val innerApply = treeCopy.Apply(tree, doWhileLoop, List(body))
          val outerApply = treeCopy.Apply(tree, innerApply, List(cond))
          outerApply

        case _ => super.transform(tree)
      }
    }
  }
}
```

Figure 6.1: The looprecover phase



(a) Original AST with goto's

(b) New AST with call to `whileLoop`

Figure 6.2: Recovering a `while` loop

6.1.3 The Global class

There is one very important class in the whole compiler: the `nsc.Global` class. Put roughly, it *is* the compiler. It contains everything, whether by mixin composition (for the symbol table, for example) or internal objects (e.g. for the phases).

Virtually all other classes in the compiler have a reference to their driving `Global` instance, under the `val global` instance value. Moreover, many of them begin their definition with `import global._`, thereby importing all definitions of the global instance into the local namespace.

It is `Global`'s role to define the set of so-called *internal phases*, i.e. phases that are part of the original Scala compiler. There are also platform phases (e.g. `jvm`) and plugin phases (brought into the compiler by plugins). This is the entry point of Ozma specialization: redefine the set of internal phases, and define a specific platform.

6.2 Oz encoding scheme

One of the key design issues was how to encode, at the macro level, Java-like classes into Oz classes. Indeed, `nsc` focuses its efforts towards classes targeted at Java. But Oz classes are very different from Java classes.

For one thing, there is no such thing as an interface in Oz. Besides, objects do not know their class. Actually Oz encourages duck typing[1]. So there is no equivalent to `isInstanceOf` or `asInstanceOf`, nor is there something like `getClass`. In fact, classes in Oz are only a means of defining how an object will behave, not what types the object conforms to.

Another issue is that in Java, classes are automatically loaded as needed, and we know where to find them thanks to two things: their fully qualified name, and the global classpath. Oz requires that modules be imported by their definite URL. This seems to be at the other end of the spectrum.

A third major issue is that Oz does not support any kind of method overloading, which is extensively used in both Java and Scala.

[1] See http://en.wikipedia.org/wiki/Duck_typing.

The last big issue we will consider is the implementation of *arrays*. In Java, arrays are instances of special classes, that are created automatically as needed, at runtime. But only one array class is created for all arrays in the VM that have the same element type. And the class of an array has means to know its element type, which allows for `instanceof` tests that also test the element type of the array.

This section will present the solutions we have brought to these issues.

6.2.1 Encoding classes in Oz

Classes are not run-time data structures in Java. All run-time manipulation of classes is done by objects that represent the classes. These objects are sometimes called *run-time classes*, and they are instances of the class `Class`. A run-time class has information about the methods of the class and about the class hierarchy. It therefore stores all the run-time classes (i.e., objects) that are its ancestors. Any object `obj` contains a reference to its run-time class, which can be obtained by `obj.getClass()`.

What about the run-time class, call it `objc`, that represents the class `Class`? A call to `objc.getClass()` will get a reference to the run-time class that represents `Class`, that is, itself. Therefore this is a circular data structure. The question is, how do we create this circular data structure? There is a second loop since `objc` also contains a reference to the run-time class representing its superclass, namely the object representing `Object`. This second loop will be handled the same way as the first. This is shown in figure 6.3.

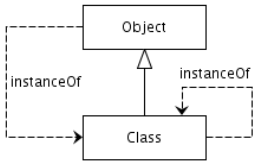


Figure 6.3: Core meta-object protocol of Java

Creating `objc` requires no infinite calculation. There is only one run-time class representing the class `Class` and one run-time class representing the class `Object`. There are two run-time classes in all: two objects, and the maximum possible calculation that will be done is to create both of them. The only question is, when do we create them, since in order to create them they must already exist (they are passed in as arguments to their constructors). This can be solved easily in Oz: create an unbound variable for each run-time class, and use it as if it were the run-time class. It can be passed as an argument to any method that requires a reference to the run-time class. It will be bound to the run-time class once. The only possible problem is if the run-time class needs to be called in order to bind the variable to the run-time class, since that will create a deadlock. This is not a problem, because it is not necessary to call the run-time class in order to create it (it is simply stored inside the object).

With run-time classes, it is easy to implement `isInstanceOf` and `asInstanceOf`, since all objects have a reference to their run-time class (accessible with `obj.getClass()`) and run-time classes have references to all their ancestors. `obj.isInstanceOf[SomeClass]` looks up `SomeClass` in the inheritance list of `obj.getClass()`. `asInstanceOf` is then trivially implemented in terms of `isInstanceOf`.

6.2.2 Loading classes by relying only on their name

The second issue was that Oz module loading requires a functor to specify the full URL to the "physical" location of the `.ozf` files containing the compiled functors it imports. In Java, the VM is able to search in several locations for a file matching the full name of the class, by using a global classpath.

Fortunately, there is a hidden feature of the standard library of Oz that allows to implement that: URL resolvers. Well, URL resolvers are not themselves hidden[2], but what is really hidden is that the functor importing system uses one, and that it is actually possible to replace it with a custom one!

Thanks to this feature, we can implement the classpath system as follows. First we use Java's conventions to send every compiled class in a directory matching its enclosing package, in a file corresponding to its name. The difference is that we group together all classes that have the same basename into a single functor with this basename. The basename of a class is the name of the top-level enclosing class.

We do that because there are often lots of interactions between nested classes and their enclosing class. So anyway, when one is loaded, the other would be loaded as well. Grouping them in a single functor reduces the number of loading operations, while keeping the binary directories a little cleaner than with Java. The phenomenon of nested classes is amplified by Scala, because it turns every lambda expression into a separate nested class (see section A.3.1).

The following step is to define a URL scheme for these functors. A functor with basename `pack.subpack.TheClass` has URL `x-ozma://root/pack/subpack/TheClass.ozf`. We need the `root` part because Oz URL resolver insists on converting the first path item to all-lowercase, hence breaking when trying to load a class in the default package.

Finally, the runtime engine has to set up a custom URL resolver based on the classpath (options `--classpath` or `-p` as well as `--bootclasspath`). Figure 6.4 shows the relevant parts of the Ozma engine, coming from `src/engine/OzmaEngine.oz`.

The `MakeResolveHandlers` basically splits a classpath string following the colons, and creates a resolver using `Resolve.handler.prefix` for each substring. It returns a list of the created resolvers. The main program calls this method for the three classpaths used by Ozma (system, boot and regular classpaths). It concatenates the resulting resolver lists, and passes the result to the `Resolve.pickle.setHandlers` function, in order to override the resolvers used by the importing system.

6.2.3 Overloading

Java and Scala make extensive use of overloading. However Oz does not support this feature. In Oz, the method label is the only thing that identifies a method. This means that we cannot rely only on the name of a Java method when we give a name to the Oz method.

In order to support overloading in Ozma, we define how to name a method for its integration in Oz. Obviously, we want the Oz name to be identical for two methods iff Java would consider these two methods to be identical. This asks the question: what are the criteria for Java to match two methods?

Java defines two methods as being *overriding-equivalent* (i.e. non overloading) iff they have the same *erased signature* [Jav05, sec. 8.4.2]. The erased signature is equivalent to the signature whose parameterized types have been replaced by their corresponding raw type.[3] The signature includes the name of the method, the types of the parameters, but *not* the return type.

[2] See http://www.mozart-oz.org/documentation/system/node50.html.

[3] This does not apply to array types, as they are not parameterized types in Java.

```
% Ozma engine - main program
% @author Sebastien Doeraene
% @version 1.0
functor

import
   System Module Application Resolve

define

   fun {MakeResolveHandlers Prefix ClassPath}
      case ClassPath of nil then
         nil
      else
         Path Tail
      in
         {String.token ClassPath &: Path Tail}
         {MakeResolveHandler Prefix Path}|{MakeResolveHandlers Prefix Tail}
      end
   end

   fun {MakeResolveHandler Prefix Path}
      {Resolve.handler.prefix Prefix Path}
   end

   try
      % Parse command-line arguments
      Args = {Application.getArgs OptSpecs}
   in
      % Set up classpath
      local
         Sys = "x-ozma://system/"
         Root = "x-ozma://root/"
         SystemHandlers = {MakeResolveHandlers Sys Args.systempath}
         BootHandlers = {MakeResolveHandlers Root Args.bootclasspath}
         Handlers = {MakeResolveHandlers Root Args.classpath}
         AllHandlers = {Append SystemHandlers
                        {Append Handlers BootHandlers}}
      in
         {Resolve.pickle.setHandlers AllHandlers}
      end

   catch error(ap(usage Message) debug:_) then
      {System.showError Message}
      {ShowUsage 1}
   [] error(Err debug:d(info:Info stack:Stack)) then
      % ...
   end

end
```

Figure 6.4: Setting up a custom URL resolver for classpath handling

```
def paramsHash(paramTypeNames: List[String], resultTypeName: String): Int =
  paramsHash(paramTypeNames ::: List(resultTypeName))

def paramsHash(paramAndResultTypeNames: List[String]) = {
  paramAndResultTypeNames.view.zipWithIndex.foldLeft(0) {
    case (prev, (item, idx)) => prev ^ Integer.rotateLeft(item.##, idx)
  }
}
```

Figure 6.5: Algorithm for the computation of the signature hash

However, this specification applies at the *user level*, to specify whether a textual method overrides another one or not. But internally, when two methods are override-equivalent but have different return types, the compiler generates a *bridge* method that has the same return type as the inherited method, and that calls the overriding method.

Therefore, *internally*, two methods are override-equivalent if they have the same name, the same number and types of parameters, and the same return type. Because our implementation must mimic the JVM behavior, we must take all these elements into account when deriving the Oz name of a method.

We build the Oz name of a method as a concatenation of the Java name with a hash of the parameter and return types.[4] The hash is defined algorithmically. The code is given in figure 6.5. In a nutshell, we compute the individual hash-code of the full names of the types. Basically we xor all these hashes. But to ensure that two methods with the same types, but in a different order, have different hashes, we inject a bit rotation of every hash following its index in the type list.

This definition obviously guarantees that override-equivalent methods have the same hash. But as is the case with every hash, there is no guarantee whatsoever that two methods that are *not* override-equivalent will indeed get different hashes. In other words, there is no guarantee there is no *collision*. This hash was designed so that it is not obviously wrong, but it is not quite correct either. If the hierarchy of a class were to contain a hash collision, the runtime engine would consider there is overriding when it should be overloading. Hence the semantics will be broken. This ought to be improved.

6.2.4 Arrays

The last implementation detail we will discuss is how to implement Java arrays in Oz. The really tricky thing about arrays is that all arrays do not share a unique class `java.lang.Array`. Instead, a separate array class is built by the JVM for every distinct element type. However, the specification guarantees that any two array classes of the same element type will be the same class.

In order to implement this, we need a cache of already created array classes, indexed by element type. That could of course be implemented as a global hash table, but there is much a better implementation: each class object has a field that contains the reference to the corresponding array class.

This also applies to array classes themselves, in order to support multi-dimensional arrays. This forms infite series of classes (`Int`, array of `Int`, array of array of `Int`...). We can avoid infite loops by creating array classes *on-demand*, using lazy execution (`byNeed`).

[4] For constructors, the return type is considered to be the enclosing class, not `Unit`.

All the array classes share the same implementation, i.e. the same Oz class. It can be found in `src/javalib/java/lang/Array.oz`. Only the instances of `Class` are different for each array type. Conveniently, our encoding scheme, which separates the `Class` instance from the Oz class, adapts very well to this implementation design: we can use a single Oz class for many `Class` instances.

Interrestingly, this design also allows for different Oz classes for different instances of the same Java class. This is also used here, as we use a different implementation of `java.lang.Class` for instances that represent array types. This alternate implementation is found in the same file, under the Oz class `ArrayClassType`.

There is another subtle issue. Upon instantiation, a Java array is required to have all its fields initialized to "the zero" of its element type. What the zero is depends on the type. For integers, it is the value 0. For objects, it is `null`. For booleans, it is `false`, and so on. Since all array classes share the same implementation, how do we adequately initialize the array contents?

In order to support this, we add another method in `Class`: `zeroOfThisClass`. This method returns `null` for all instances of `Class` corresponding to classes. It returns a specialized value for instances of `Class` that represent primitive types. Again, we use a separate implementation of the class `Class` for primitive types. It is found in `Class.oz` under the name `Class$PrimitiveClass`.

6.3 The Ozma compiler

This section will present the major architecture and design choices of the Ozma compiler. As we have already mentioned, it is designed as an extension to `nsc`, that modifies the set of internal phases as well as the platform. In order to do so, we extend the `Global` class with the *trait* `OzmaGlobal` (in package `nsc.ozma`).

We use a trait instead of a subclass because `Global` already has a subclass, `interactive.Global`, that supports the interactive environment of Scala. We designed `OzmaGlobal` as a trait in the hope that, in the future, we will have an interactive environment for Ozma. Then we can compose `OzmaGlobal` with `interactive.Global` to obtain the Ozma interactive compiler.

6.3.1 The OzmaGlobal trait

The `OzmaGlobal` trait makes all the amendments to `Global` that make up the Ozma compiler. It has to:

- Replace the back-end (JVM or MSIL) by the Mozart back-end,
- Provide a composite object for handling Oz code (replacing the one that handles icode in the Scala compiler),
- Define the phase objects (instances of `SubComponent`) that are specific to Ozma (added or overridden) and
- Override the set of so-called *internal phases*, in order to discard the phases not needed for Ozma and add the new ones.

Figure 6.6 shows a clipped view of `OzmaGlobal`. It extends `Global`, obviously, and `OzmaTrees`.[5]

On line 8, we override the platform (JVM or MSIL in `Global`) with a custom platform, `MozartPlatform` (in package `nsc.backend`). The platform specifies what are the platform phases (here `mozart` only).

[5] In case you wonder, `OzmaTrees` is a tiny, non interresting, implementation detail.

```
package scala.tools.nsc
package ozma

// imports

trait OzmaGlobal extends Global with OzmaTrees {
  /** Platform */
  override lazy val platform: ThisPlatform =
    new { val global: OzmaGlobal.this.type = OzmaGlobal.this } with MozartPlatform

  /** OzCode generator */
  object ozcodes extends {
    val global: OzmaGlobal.this.type = OzmaGlobal.this
  } with OzCodes

  // OZMA Compiler phases ------------------------------------------------------

  // phaseName = "looprecover"
  object whileLoopRecovering extends {
    val global: OzmaGlobal.this.type = OzmaGlobal.this
    val runsAfter = List[String]("parser")
    val runsRightAfter = None
    override val runsBefore = List[String]("namer")
  } with WhileLoopRecovering

  // [...] Other phase definitions

  /** Add the internal compiler phases to the phases set.
   *  This implementation creates a description map at the same time.
   */
  override protected def computeInternalPhases() {
    // Note: this fits -Xshow-phases into 80 column width, which it is
    // desirable to preserve.
    val phs = List(
      syntaxAnalyzer          -> "parse source into ASTs, perform simple ...",
      singleAssignVals        -> "take care of single assignment values",
      whileLoopRecovering     -> "recover while loops",
      analyzer.namerFactory   -> "resolve names, attach symbols to named trees",
      analyzer.packageObjects -> "load package objects",
      analyzer.typerFactory   -> "the meat and potatoes: type the trees",
      superAccessors          -> "add super accessors in traits and ...",
      pickler                 -> "serialize symbol tables",
      refchecks               -> "reference/override checking, translate ...",
      uncurry                 -> "uncurry, translate function values to ...",
      ozmaSpecializeTypes     -> "@specialized-driven class and method ...",
      ozmaExplicitOuter       -> "this refs to outer pointers, translate ...",
      erasure                 -> "erase types, add interfaces for traits",
      lazyVals                -> "allocate bitmaps, translate lazy vals ...",
      ozmaLambdaLift          -> "move nested functions to top level",
      constructors            -> "move field definitions into constructors",
      mixer                   -> "mixin composition",
      ozcode                  -> "generate Oz code from AST",
      tailcalls               -> "rewrite tail calls",
      ozmaTerminal            -> "The last phase in the compiler chain"
    )

    phs foreach (addToPhasesSet _).tupled
  }
}
```

Figure 6.6: The `OzmaGlobal` trait

Line 12 introduces the `ozcodes` object, of class `nsc.backend.ozcode.OzCodes`. This class defines all the methods and classes that are useful to work with Oz ASTs. It has a role similar to the `icodes` object of `Global` for icode manipulation.[6]

After that come the definitions of several objects for the subcomponents of `OzmaGlobal`. Here we show only one such component, the one that corresponds to the `looprecover` phase that we presented earlier. Note that each phase declares dependancy constraints with other phases. `looprecover` declares that it must be run after `parser` but before `namer`. The `nsc.PhaseAssembly` trait is responsible for solving the set of constraints declared by all phases in order to define a linear order [Bac08].

Finally, the `computeInternalPhases` method (line 31) overrides the set of internal phases specified in `Global`.

6.3.2 GenOzCode: generating Oz code from the AST

Probably the most important class in the Ozma compiler is `nsc.backend.mozart.GenOzCode`. It is responsible for converting a Scala AST into Oz code. It outputs a structure that we call *Oz code* (analogous to *icode* in Scala), which is a set of instances of `OzClass`, defined in `ozcode.Members`. Each `OzClass` contains a set of fields and methods (`OzField` and `OzMethod`). Finally, each method contains its body as an Oz AST fragment matching a phrase.

An Oz AST is a tree of case classes defined in `backend.ozcode.ASTs`. Its structure follows directly from the ASTs used internally by the Mozart compiler, and documented in [Kor].

This design is similar to the one used to store icode. In this respect, it replaces the `icode` phase used by `nsc`, and the corresponding phase is thus named `ozcode`.

We will not go through the entire code of `GenOzCode`. This would be both too long and useless. It is sufficient to know that the most important method is `genExpression`. This is the base method for translating a Scala expression AST into an Oz expression AST, i.e. a phrase. Remember that statements are simply expressions that return `Unit`. The interrested reader can browse the code by himself.

6.3.3 GenMozart: from Oz code to complete Oz functors

The `GenMozart` class is the last phase in Ozma. It takes Oz code (i.e. a set of `OzClass`) as input and outputs compiled Oz functors. It works in several steps.

1. Produce the ASTs for all the Oz classes, one for each `OzClass`,
2. Produce lazy-executing global variables: modules and runtime classes,
3. Group them all in several functors according to their basename,
4. Detect the imports and exports appropriate to each functor,
5. Build the complete AST of each functor,
6. Compile this Oz AST into an `.ozf` file by calling a modified Oz compiler.

We modified the Oz compiler so that we can give it directly the AST of the functor instead of the corresponding source code. This brings one major advantage: we keep source code position annotations right through the Oz compiler.

[6] The definition of `val global` inside the refinement is a common pattern in `nsc`: classes are defined abstract, with an abstract `val global: Global`. In `Global`, the class is instantiated with a refinement that binds `global` to `this` instance of `Global`.

6.3.4 Natives: producing native methods

Several methods in the standard libraries of Java, Scala, and Ozma are annotated with `@native`. Such methods cannot be defined in Scala code because they are too low-level: the compiler (or an external means, e.g. for JNI) must know "by heart" what the output code is for those methods.

In order to support these, there is a class, `Natives`, that is responsible for defining the output Oz code of such methods. In this class, there is a nested `object` for each native method supported by Ozma. These objects extend `NativeMethod`, an abstract class with an abstract `body` method. This method is defined in every object and returns directly the AST of the method code.

Because writing raw ASTs in every `body` method is tedious and unreadable, there is a helper nested object, called `astDSL`. This object defines a DSL (domain-specific language) in Scala to represent Oz code.

6.4 Dataflow values

Now that we have covered the general design choices as well as the architecture of the Ozma compiler, we will finally focus on the implementation of dataflow variables. This feature impacts the compiler at several phases, because it fundamentally changes the semantics of what a `val` is.

Basically, it is `GenOzCode`'s job to take care of the different semantics of assignments, for dataflow values and for variables. When compiling an assignment, it will look at the symbol flags of the left-hand side (lhs) of the equals. If it is mutable (i.e. a `var`), it will compile the assignment as an Oz assignment. Otherwise, it is a `val`, and an Oz *binding* operation is produced.

When encountering a `val` in a right-hand side (rhs), we simply access it. Whereas if we encounter a mutable symbol, we dereference the cell using the `@` operator.

The following code snippet illustrates the various cases:

```
val value = 5
var variable = value + 3
variable = variable * value
```

This is compiled as the following Oz code:

```
local
   `value~10201`      % the numbers are internal IDs
   `variable~10200`   % that are likely to change
   `singleAss~10199`
in
   `variable~10200` = {NewCell 5}
   `value~10201` = @`variable~10200` + 3
   `singleAss~10199` = @`variable~10200`
   `variable~10200` := @`variable~10200` * `value~10201` + `singleAss~10199`
end
```

This is sufficient for all regular `val`'s and `var`'s. But there is a major issue: the initially unbound values. For one thing, it is syntactically illegal in Scala to declare a local `val` without an initial value. Then, it is illegal to assign something to a value.

Actually, for most of the compiler front-end, a single-assignment value must look like it is a variable for the compiler to do its job correctly.

6.4.1 The internal `@singleAssignment` annotation

The basic idea is to transform, in the Scala AST, initially unbound values as local *variables*, with a dedicated annotation `@singleAssignment`. We have to take care of that very early in the compilation chain, namely before the `namer` phase (which is second in the Scala compiler).

Indeed, it is `namer` that checks for uninitialized local values. So we define a new phase, `singleass`, that will be run after `parser` but before `namer` (as was `looprecover`, actually).

This phase identifies local *deferred* (Scala word for abstract) non-mutable values, and will transform them into local mutable values annotated with `@singleAssignment`. Additionally, they are given a dummy initialization expression.

The code for this phase can be found in the class `nsc.ozma.SingleAssignVals`. It is relatively simple, and will therefore not be discussed in detail.

The `ozcode` phase takes `@singleAssignment` annotations into account, and treats all variables that hold this annotation as if they were values, which, actually, they *are*.

This simple transformation solves *all* the issues introduced by single-assignment values. But unfortunately, it also itself introduces a new issue.

6.4.2 Dataflow values captured by `lambdalift`

The `lambdalift` phase is an existing phase of the Scala compiler. It is responsible for unnesting all nested classes and functions. While doing that, it creates additional class fields and function parameters to *capture* free values and variables. A very good explanation of the needs, issues, and algorithms of this phase can be found in [Alt06, cpt. 3]. A less theoretical approach can be found in [Gar11d].

The problem with `lambdalift`, from the point of view of Ozma, is that it was not designed with dataflow values in mind (obviously), and it does a very bad job with single-assignment values.

The main issue is that, for this particular phase, we want single-assignment values to be actually considered as values. So we replace the standard `lambdalift` phase by a custom one, that extends it, and is specialized for Ozma. We redefine the `apply` method with this simple code:

```
override def apply(unit: global.CompilationUnit) {
  convertSingleAssignVars(unit, before = true)
  super.apply(unit)
  convertSingleAssignVars(unit, before = false)
}

def convertSingleAssignVars(unit: CompilationUnit, before: Boolean) {
  val singleAssignment = definitions.getClass("scala.ozma.singleAssignment")
  for (tree @ ValDef(_, _, _, _) <- unit.body;
       if (tree.symbol.hasAnnotation(singleAssignment))) {
    if (before)
      tree.symbol.resetFlag(MUTABLE)
    else
      tree.symbol.setFlag(MUTABLE)
  }
}
```

This turns `@singleAssignment`-annotated symbols back to immutable before to apply the inherited algorithm, and turns them back as mutable after. Crude, but effective.

The problem is that, it does not solve everything. There is a more subtle issue left: the algorithm does not replicate annotations from the local values it captures to the class fields it creates. So, arriving at `ozcode`, we have wrong information. If the methods defined in the `LambdaLift` class had not all been private, we could have changed that by extension. But unfortunately it is not the case.

So the idea, to solve the issue, is to apply a post-processing step to the job done by the standard `lambdalift` phase, at the end of `apply`. This step scans the resulting AST, and identifies class fields that were created by `lambdalift` from single-assignment values. It then adds the `@singleAssignment` annotation to them.

How we achieve that is a bit technical and not worth the details. Basically we identify constructor calls to synthetic (compiler-generated) classes. For all actual parameters that are single-assignment values, we search for a field in the class with the same name. If this field is also synthetic, and is a so-called param-accessor, then it has been generated by `lambdalift`, and we annotate it with `@singleAssignment`.

If single-assignment values ever make their way through the official Scala compiler, all this mess could be rewritten much more cleanly, with a flag instead of an annotation, and with dedicated code inside the `LambdaLift` class.

6.5 The `@tailcall` annotation and the `tailcalls` phase

The last implementation point that we will consider in this text is the implementation of the specifications of the `@tailcall` annotation. This annotation applies to any method or constructor parameter, and is the basic building block for advanced tail call optimization based on dataflow values.

In order to implement the semantics defined in section 5.6, we add a dedicated phase, `tailcalls`, that takes place after `ozcode` but before `mozart`. Thus, it works on Oz code. The code is available in class `nsc.backend.ozcode.opt.TailCalls`.

Since this phase works on Oz code, we need a means to retain `@tailcall` information within Oz code ASTs. We store in each `Apply`-like node a list of the indices of `@tailcall`-annotated parameters, in priority order, i.e. in *right-to-left* order. This information is encapsulated in the `ast.TailCallInfo` trait.

`TailCalls` applies, for each method of each class, the following steps. It first looks for any tail call that should be transformed. If it does not find any, the phase does not transform the method at all.

Otherwise, it first rewrites the method so that it has an explicit result value, i.e. an output parameter. It then looks again for tail calls, and transforms them as they are reached.

We will not discuss the matter further, as the actual algorithm is straightforward from the semantics specifications.

6.6 Evaluation

This last section evaluates the current implementation of the compiler.

6.6.1 Coverage of the semantics of Scala

Thanks to the great modularity of the Scala compiler, we were able to reuse all the parts that we needed in order to support the specifications of Scala [Ode11]. Therefore, our compiler covers the entire specification.

6.6.2 Correctness of the compiler

To our knowledge (i.e. unless there is a bug), the only possible flaw in the Ozma compiler is concerned by *jumps*. As we already mentioned, the internal ASTs of Scala can contain jumps, because of a) loops and b) `match` statements. We take care of the former with the `looprecover` phase. In order to support `match` statements, we have written a partial translation function for jumps. We have no proof that it works for all ASTs produced by `match` expressions, but we could not find a counter-example (even through the entire Scala library).

We tested the compiler for correctness and absence of memory leaks on all examples of this dissertation as well as example programs accompanying the source code. Also, we tested it on the non-trivial program Capture the Flag. And last but not least, we used it to compile the entire Scala library, that is used by all these examples.

6.6.3 Coverage of the Java library

The most important gap in our implementation is the Java runtime library. We translated only the core classes up to now. This poses severe restrictions on the code that can be used. For example, collection methods on arrays are limited by the fact that we do not support reflection-based array creation in our Java library.

As will be discussed in chapter 7, this should be addressed soon.

6.6.4 Coverage of Oz concurrency features

Ozma supports most concurrency features of Oz: dataflow concurrency and message-passing concurrency were the focus of this work. Shared state concurrency exists in Scala, so it exists also in Ozma.

There are essentially two aspects that we lose in Ozma: full unification and distributed programming.

As was discussed in section 3.1.2, Ozma only supports shallow unification. We made this trade-off in order to keep all object-orientation of Scala intact.

As for distributed programming, this could be a major field of research following this work, as we discuss in section 7.2.

6.6.5 Performance

The code produced by Ozma performs awfully, compared to Scala and Oz, though in absolute it has acceptable performance. This is a purely subjective analysis, as we did not run any performance test.

This work was focused on provided a maximum of functionality, not on achieving good performance. However, we have already identified potentials for optimizations, that are discussed in section 7.3.

Chapter 7

Further work

The two previous chapters have described the design and implementation of Ozma. This chapter covers the possible improvements for the future of Ozma. There are possible paths that we can take from now on. Some short term requirements to make Ozma fully usable. And longer term possibilities to make it an important alternative to both Scala and Oz.

7.1 Possible chronology

During this work, we focused on theoretical aspects of Ozma features, as well as their implementation. But we left out some implementation details of practical importance to make Ozma a useful tool.

7.1.1 Short term

In the short term, we ought to address the following issues:

Separate compilation of Ozma programs

Though it is possible to compile several source files together to form an application, the actual implementation cannot cope with Ozma libraries compiled separately. The standard Scala library, as well as the standard Ozma library, are sort of special-cased to support compilation of an application separately from these. But it is impossible to separately compile a user-defined library and an application using it.

The reason for this is that type signatures of classes are not stored along with their compiled Oz functors. Hence, we cannot reload them without the source code. The official Scala compiler stores its type signatures in Java annotations [Dub10]. In Ozma, we should design a similar mechanism for storing signatures in Oz functors, or in a file beside them if the former reveals to be unfeasible.

Extend the available standard Java library

Scala relies on the Java standard library for a lot of core tasks. The `Object`, `String` and `System` classes are trivial examples, but there are many of them. Although the Ozma compiler is able to compile existing Scala code, it cannot compile Java classes directly. In the present implementation, only the core classes of the Java library have been rewritten in Scala/Ozma so that they can be compiled to Oz functors.

In the near future, we should provide wider access to the Java library. Existing tools that rewrite Java source files to Scala, such as Jatran[1], could be helpful in this task.

In the longer term, we could develop a Java front-end to Ozma, thereby enabling to directly compile any Java class for use by Ozma.

Fault-tolerant distributed programming from Oz

See section 7.2.

7.1.2 Middle term

In the middle term, we should consider the following improvements.

Optimizations

See section 7.3.

Interactive environment

Scala and Oz both provide interactive environments. Oz through the OPI, and Scala through its interactive interpreter. Ozma should also support such a tool.

This is a challenge for Ozma, as it is built from two separate languages. In both Scala and Oz, the interpreter and compiler are written in the same language, and run on the same engine. This allows relatively easy integration of both. Ozma, however, has a compiler running on top of the JVM, and an interpreter running of top of the Mozart engine. It is therefore not straightforward at all to mix the compiler and interpreter.

7.1.3 Long term

In the long term, it is possible that we tackle the following opportunities.

Self-compilation

The Ozma compiler is written in Scala, now it is supposed to be able to compile any Scala code, since it is a conservative extension to Scala. When the compiler will be mature enough, and the generated code sufficiently optimized, it will be a major proof of maturity to make it compile itself. The resulting compiler would then run on top of the Mozart engine.

Upstream implications

See section 7.4.

7.2 Fault-tolerant distributed programming from Oz

One of the great strengths of Oz is its distributed model. Oz provides network transparent distribution inside the language, with fault streams [Col07]. This model is an extension of Erlang for temporary failures.

Additionaly, there is work in progress to make this *scalable* [JV11]. In the era of Internet and cloud computing, this is becoming more and more important.

In the short term, these models should be fairly easily imported in Ozma, as we use the same runtime environment. Care must be taken, though, because these systems are most efficient in

[1] http://code.google.com/p/jatran/

the presence of declarative data structures. In Oz the immutability of data structures in enforced by the language, e.g. for records. In Ozma all data structures are objects, which the runtime cannot consider as immutable. This extension is thus not trivial.

In the long term, we expect the integration of distributed programming in Ozma to become more important. The subject is still open in Oz, so a fortiori there will be a lot of work to undertake in Ozma.

7.3 Optimizations

In this work, we have made a priority to have a working compiler, that supported the entire Scala specification. We did not care about the efficiency of the compiled program. Efficiency of the compiler itself is not an issue. The Scala front-end that we reuse is already the bottleneck of compilation time.

The current implementation of the Ozma compiler is mainly located in the back-end of the compiler. We reuse almost all the front-end of the Scala compiler, which is designed to produce Java-like classes. However, Oz has more concepts in common with Scala, e.g. first-class functions. This offers some opportunities for major optimizations.

7.3.1 Compile Function values as Oz functions instead of objects

Scala supports first-class functions, while Java does not. To achieve this, the Scala compiler rewrites lambda expressions to anonymous classes implementing one of the `FunctionN` traits. Then it extracts these nested anonymous classes to the top-level.

Now, in Oz, first-class functions do exist, up to the runtime, and such first-class functions are a lot more efficient than object manipulation. Therefore, we would like the Ozma compiler to keep lambda expressions as is and compile them as such in Oz.

At first sight, this seems to be easy: there is simply less work to do. But it is not that simple. The problem is that the `FunctionN` traits are not only an implementation detail. They are part of the specifications. We can write user-defined classes that extend a `FunctionN` trait and implement its `apply` method as we want. If we compile lambda expressions as Oz anonymous functions, we need to make sure that such classes can be automatically converted to Oz anonymous functions as well when assigned to a value of type `FunctionN`. Worse, we need to make sure that we can call standard methods on any `FunctionN` value, including `asInstanceOf` for downcasting purpose, which implies that we must be able to recover the original instance from the function.

The main idea to solve this problem would be to encode `FunctionN` values as Oz *pairs* of (a) an Oz function and (b) a Scala object. Depending on the source (a lambda expression or an instance), one of the fields would be determined, and the other one a lazy value computed from the first one. When calling the function, we use the first field. For any other operation, we use the second field. Convertion would happen on-demand through the lazy field.

7.3.2 Compile Ozma lists (and case classes) as actual Oz lists and records

Another opportunity for optimization are lists, are more generally case classes. Oz has the concept of *records*, which are basically very limited case classes. They have only immutable fields. Besides, operations on lists are optimized by the Oz back-end and engine, and there is a `case` statement in Oz that corresponds to `match` in Scala for records.

In some cases, we could compile Ozma case classes as Oz records, and `match` statements as Oz `case` statements.

This raises many issues similar to those of first-class functions, yet still worse. What happens to their methods? How do we manage inheritance with those? These are non-trivial questions that we have not thought about yet.

7.4 Upstream implications

The global hope of this work, for the future of programming, is that we can show to the Java and Scala communities how useful are dataflow variables and other concepts of Oz. We hope that Ozma will be the first step to introduce these concepts in mainstream languages.

7.4.1 A modified JVM to support dataflow values

The most serious perspective would be to bring dataflow values to the official Scala language. Dataflow values are the core of most the concepts introduced by Ozma. The main obstacle is that dataflow values as we know them require support from the runtime engine.

Hence, we would need to modify the Java Virtual Machine so that it supports dataflow values. There is no technical issue about that: Flow Java [DSHB03] has already given the path towards single assignment values in the JVM. But unfortunately, the Java people are not quite ready for this, and will resist strongly unless we can prove them that it is a very useful improvement.

Chapter 8

Conclusion

The main goal of this dissertation was to design a conservative extension of Scala with concurrency features coming from Oz, and implement it. In order to achieve this, we have developed a compiler, based on the Scala compiler with a dedicated back-end, and a runtime environment, based on the Mozart programming system.

We have succeeded in designing the language Ozma so that it includes the core concurrency features of Oz (dataflow variables, light-weight threads, lazy execution and ports) while preserving the existing semantics of Scala. We also succeeded in developing a working compiler for this language. As a proof of concept, we have been able to compile and use the entire Scala standard library in Ozma program, while injecting dataflow values in standard collections, and got the expected results.

This has been demonstrated by the list-based examples we have shown in Ozma, and also by the example programs we have provided. The most important proof of expressivity is the implementation of Capture the Flag (section 4.3.4). Indeed, it shows that all the programming techniques taught in the Oz course at the Université Catholique de Louvain are supported by Ozma.

There are however two issues that we need to consider: some flaws in the runtime environment, and the trade-offs we applied on Oz features.

The runtime environment of Ozma must basically mimic the JVM behavior, hence follow the Java specifications [Jav05]. It has three major components:

- The Mozart runtime engine,
- The Ozma runtime engine (found in `src/engine/`), and
- The emulation of the Java standard library (found in `src/javalib/`).

We believe the Ozma runtime engine to be correct with respect to the specifications of Ozma. However, both the Mozart runtime engine and our emulation of the Java standard library have flaws with respect to these specifications.

First, the Mozart engine has some core features that are designed for Oz, but that are inconsistent with the core specifications of Java, hence Scala and Ozma. The best example are integers.

In Oz, all integers share the same type, and are *big integers* by nature. Whereas in Java, there are four integer types, each one with a specific domain range, and arithmetic operations are required to conserve the domain range (which implies truncations at some point). Though

theoretically undesirable for reasoning easily about programs, this specification is relied upon by some algorithms in Java and Scala code (e.g. hashcode computation). There is no way, to our knowledge, that we can make Mozart behave this way.

The other major example is builtin exceptions. In Java, dividing a primitive integer by 0 is supposed to throw an `ArithmeticException`. Doing so in Mozart will also raise an exception, but a Mozart one (a so-called kernel error with message `'div0'`). Concretely, in the current implementation of Ozma, this means that an exception handler expecting an `ArithmeticException` will be by-passed.

The emulation of the Java standard library is only minimal at this stage. This means that most applications using non-core functionalities of this library will not run properly. As was said in section 7.1.1, this should be fixed soon.

We must also consider the trade-offs we made in order to fit Oz features into Scala. A lot of Oz idioms use *records* (lists are a particular case). In Ozma we have not attempted to use them. We have followed Scala in its philosophy that everything is an object (including primitive data types, from a conceptual point of view). This leads to major inefficiencies compared to a similar program written in Oz. This was motivated by the *conservative* aspect of our extension of Scala. We believe though that this trade-off could be eliminated by appropriate optimizations.

We have also dropped some advanced features of Oz, like *structural unification*. We believe that this is not a problem for everyday programming, because the full power of structural unification is rarely needed. This does nevertheless limit the expressivity, compared to that of Oz.

All in all, the Ozma language is a practical extension to Scala, offering the core concurrency features of Oz. But it is not mature yet, and there is plenty of room for improving it.

Bibliography

[Alt06] Philippe Altherr. *A Typed Intermediate Language and Algorithms for Compiling Scala by Successive Rewritings.* PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 2006.

[Arm03] Joe Armstrong. *Making reliable distributed systems in the presence of software errors.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, Dec. 2003.

[Bac08] Anders Bach Nielsen. Scala compiler phase and plug-in initialization for Scala 2.8. Scala SID document, 2008.

[Bou03] Clara Bouzas Manté. Typed-oz. Master's thesis, Université Catholique de Louvain, Belgium, 2003.

[CGLO06] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In *Procedings of the 31st International Symposium on Mathematical Foundations of Computer Science*, Stará Lesná, Aug. 2006.

[Col07] Raphaël Collet. *The Limits of Network Transparency in a Distributed Programming Language.* PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, Dec. 2007.

[Cop08] Yohann Coppel. Reflecting Scala. Technical report, École Polytechnique Fédérale de Lausanne, Switzerland, Jan. 2008.

[DO09a] Iulian Dragos and Martin Odersky. Compiling generics through user-directed type specialization. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 42–47, Genova, Italy, July 2009. ACM.

[DO09b] Gilles Dubochet and Martin Odersky. Compiling structural types on the JVM. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 34–41, Genova, Italy, July 2009. ACM.

[Dra10] Iulian Dragos. Type specialization in Scala 2.8. Scala SID document, May 2010.

[DSHB03] F. Drejhammar, C. Schulte, S. Haridi, and P. Brand. Flow Java: Declarative concurrency for Java. In *Proceedings of the Nineteenth International Conference on Logic Programming*, volume 2916 of *LNCS*, pages 346–360, Mumbai, India, Dec. 2003. Springer-Verlag.

[Dub10] Gilles Dubochet. Storage of pickled Scala signatures in class files. Technical report, École Polytechnique Fédérale de Lausanne, Switzerland, June 2010.

[Emi07] Burak Emir. *Object-Oriented Pattern Matching*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, Oct. 2007.

[Gar09a] Miguel Garcia. Code walkthrough of CleanUp phase. Technical report, Hamburg University of Technology, Nov. 2009.

[Gar09b] Miguel Garcia. Code walkthrough of the UnCurry phase (Scala 2.8). Technical report, Hamburg University of Technology, Nov. 2009.

[Gar09c] Miguel Garcia. Tail self-calls become loops. Technical report, Hamburg University of Technology, Dec. 2009.

[Gar10] Miguel Garcia. Notes on genicode. Technical report, Hamburg University of Technology, Feb. 2010.

[Gar11a] Miguel Garcia. How erasure works (part a). Technical report, École Polytechnique Fédérale de Lausanne, Switzerland, May 2011.

[Gar11b] Miguel Garcia. How the constructors phase works. Technical report, École Polytechnique Fédérale de Lausanne, Switzerland, Apr. 2011.

[Gar11c] Miguel Garcia. How the flatten phase works. Technical report, École Polytechnique Fédérale de Lausanne, Switzerland, May 2011.

[Gar11d] Miguel Garcia. How the lambdalift phase works (part 1). Technical report, École Polytechnique Fédérale de Lausanne, Switzerland, May 2011.

[Gar11e] Miguel Garcia. How the lazyvals phase works (part 1). Technical report, École Polytechnique Fédérale de Lausanne, Switzerland, May 2011.

[Jav05] *Java Language Specification*. Sun Microsystems, Inc., 3rd edition, May 2005.

[JV11] Yves Jaradin and Peter Van Roy. A formal model of software rejuvenation for long-lived large-scale applications. Unpublished, March 2011.

[Kor] Leif Kornstaedt. Syntax Tree Format, Mozart documentation. http://www.mozart-oz.org/documentation/compiler/node7.html.

[LW94] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16:1811–1841, Nov. 1994.

[Med10] Kai H. Meder. ScalaFlow: Continuation based dataflow concurrency in Scala. Master's thesis, FH Wedel – University of Applied Science, Germany, Sep. 2010.

[Ode09] Martin Odersky. Compiler internals. Recorded sessions (video), 2009.

[Ode11] Martin Odersky. *The Scala Language Specification*, May 2011.

[OZ05] Martin Odersky and Matthias Zenger. Scalable component abstraction. In *Proceedings of the 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, Oct. 2005. ACM SIGPLAN.

[Ros07] Andreas Rossberg. *Typed Open Programming: A higher-order, typed approach to dynamic modularity and distribution.* PhD thesis, Universität des Saarlandes, Jan. 2007.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.

Appendix A

Phases of the compiler

The Scala/Ozma compiler is made of a set of phases. Each phase applies successive transformations of the code AST. This appendix gives an overview of each phase. Unless stated otherwise, a phase applies in both Scala and Ozma.

The option `-Xshow-phases` of the compilers (both Scala and Ozma) outputs a list of the phases that will be run, in the order in which they will be run. This order is not hard-coded in `nsc`: it is the result of a relatively constraint solving algorithm discussed in [Bac08].

A.1 Abstract Syntax Tree

Most phases (all except the back-end phases) work on a data structure that represents the Abstract Syntax Tree of the code. This data structure is made of a tree of case classes inheriting from `scala.reflect.generic.Trees.Tree`. There is a subclass of `Tree` for every syntactical element of Scala code.

In addition to its case class arguments, each node class has two important information: its *position* in the source code, and the attached `symbol`. A symbol describes all the semantic information about the node. Until the `typer` phase, AST nodes do not have symbol information.

These tree nodes are highly undocumented, though the comments at the bottom of the file `Trees.scala` provide some meaning to the node classes by giving their corresponding source code. Table A.1 show most of this information. This does not pretend to give complete understanding of the AST, but it can be used as a quick reminder.

A.2 Front-end phases

A.2.1 Parser

The `parser` phase is always the first phase to execute. It reads each source file to be compiled (a *compilation unit*), and builds the corresponding Abstract Syntax Tree. The AST produced by the parser has no symbol information.

A.2.2 `@tailcall` for case classes, Ozma only

The `casetailcalls` phase is specific to Ozma. It scans the raw AST for case class definitions. For each one, it checks whether it complies to the requirements for a case class to be automatically `@tailcall`-annotated, stated in section 5.6.

Node case class	Corresponding source code
`EmptyTree`	dummy case object representing empty subtrees
`PackageDef(pid, stats)`	`package pid { stats }`
`ClassDef(mods, name, tparams, impl)`	`mods class name[tparams] impl`
`ModuleDef(mods, name, impl)`	`mods object name impl`
`ValDef(mods, name, tpt, rhs)`	`mods val/var name: tpt = rhs`
`DefDef(mods, name, tparams, vparamss, tpt, rhs)`	`mods def name[tparams] (vparams0) ... (vparamsN): tpt = rhs`
`TypeDef(mods, name, tparams, rhs)`	`mods type name[tparams] = rhs`
`LabelDef(name, params, rhs)`	`label name(params) { rhs }`
`Import(expr, selectors)`	`import expr.{ selectors }`
`Template(parents, self, body)`	`extends parents { self => body }`
`Block(stats, expr)`	`{ stat1 ; ... ; statN ; expr }`
`ArrayValue(elemtpt, elems)`	`new elemtpt[] { elems }` (Java syntax)
`Function(vparams, body)`	`(vparams => body)` (anonymous function)
`Assign(lhs, rhs)`	`lhs = rhs`
`If(cond, thenp, elsep)`	`if (cond) thenp else elsep`
`Match(selector, cases)`	`select match { case1 ; ... ; caseN }`
`CaseDef(pat, guard, body)`	`case pat if (guard) => body`
`Return(expr)`	`return expr`
`Try(block, catches, finalizer)`	`try block catch { catches } finally finalizer`
`Throw(expr)`	`throw expr`
`New(tpt)`	`new tpt`
`Typed(expr, tpt)`	`expr: tpt` (in expressions)
`TypeApply(fun, args)`	`fun[args]` (generic instantiation)
`Apply(fun, args)`	`fun(args)` (function or method call)
`Super(qual, mix)`	`qual.super[mix]` (usually `qual` and `mix` are omitted)
`This(qual)`	`qual.this` (usually `qual` is omitted)
`Select(qualifier, name)`	`qualifier.name`
`Ident(name)`	`name` (trivial identifier)
`Literal(value)`	`value` (constant literal)

Table A.1: Abstract Syntax Tree node classes

Its implementation is quite straightforward, as it follows directly from the specification. Its source code can be found in `CaseClassTailCalls.scala`.

A.2.3 While loop recovering, Ozma only

The `looprecover` phase is extensively discussed in section 6.1.2. Its purpose is to recover `while` and `do..while` loops in the AST, that have been destroyed by the parser into label definitions and goto's. It translates them to calls to the runtime methods `whileLoop` and `doWhileLoop`.

A.2.4 Single-assignment values, Ozma only

The `singleass` phase is responsible for the syntactical amendment brought by Ozma, i.e. single-assignment values. The rationale for this phase is discussed in section 6.4.

In a nutshell, it converts every statement like

```
val value: Type
```

in a block into the statement

```
@singleAssignment var value: Type = newUnbound
```

A.2.5 Namer, package objects and typer

The three phases `namer`, `packageobjects` and `typer` are intimately related. There is no definite order between these three phases, as they call each other in non-obvious ways.

The designers of the Scala compiler themselves discourage any attempt of playing with these.

Their purpose is to annotate the entire AST with symbol and, hence, type information. After `typer`, the AST is typed and must remain so. This means that every subsequent phase must type the AST subtrees it produces.

The `typer` phase is probably the most important one in Scala. It is covered in [CGLO06].

A.2.6 Super accessors

In Scala, calls to `super` methods are resolved dynamically, since traits can be mixed in any class, at any point in the hierarchy. The `superaccessor` phase takes care of the necessary compile-time treatments. It is best understood on an example:

```
class ParentClass {
  def someMethod = 5
}

trait SomeTrait extends ParentClass {
  override def someMethod = super.someMethod + 1
}

class ChildClass extends ParentClass with SomeTrait {
  override def someMethod = super.someMethod * 2
}
```

Before running the `superaccessors`, the internal code for this little code is the following one:

```
package <empty> {
  class ParentClass extends java.lang.Object with ScalaObject {
    def this(): ParentClass = {
      ParentClass.super.this();
      ()
    };
    def someMethod: Int = 5
  };
  abstract trait SomeTrait extends ParentClass with ScalaObject {
    def /*SomeTrait*/$init$(): Unit = {
      ()
    };
    override def someMethod: Int = SomeTrait.super.someMethod.+(1)
  };
  class ChildClass extends ParentClass with SomeTrait with ScalaObject {
    def this(): ChildClass = {
      ChildClass.super.this();
      ()
    };
    override def someMethod: Int = ChildClass.super.someMethod.*(2)
  };
}
```

Compare this code with the following one, which is the result of the transformation performed by the `superaccessors` phase:

```
package <empty> {
  class ParentClass extends java.lang.Object with ScalaObject {
    def this(): ParentClass = {
      ParentClass.super.this();
      ()
    };
    def someMethod: Int = 5
  };
  abstract trait SomeTrait extends ParentClass with ScalaObject {
    private <superaccessor> def super$someMethod: Int;
    def /*SomeTrait*/$init$(): Unit = {
      ()
    };
    override def someMethod: Int = SomeTrait.this.super$someMethod.+(1)
  };
  class ChildClass extends ParentClass with SomeTrait with ScalaObject {
    def this(): ChildClass = {
      ChildClass.super.this();
      ()
    };
    override def someMethod: Int = ChildClass.super.someMethod.*(2)
  };
}
```

Note the `<superaccessor>` method in `SomeTrait`. This special method is used instead of `super.someMethod` in `SomeTrait.someMethod`. This method has no body (is abstract) and is never overridden. The `mixin` phase will define the overridden method in `ChildClass`.

A.2.7 Pickler: serialize symbol table

The `pickler` phase makes a snapshot of the symbol table, the types, the signatures, and so on. It serializes it, so that the back-end can store the Scala signature of the program in the compiled output.

Some insight about the pickler can be found in [Cop08].

A.2.8 Reference and override checking

The `refchecks` phase is a happy mix of some post-typechecking checks and totally unrelated transformations. Quoting from the ScalaDoc header of class `RefChecks`:

This phase performs the following checks:

- All overrides conform to rules (see [Ode11, sec. 5.1.4]).
- All type arguments conform to bounds (see [Ode11, sec. 3.5]).
- All type variable uses conform to variance annotations (see [Ode11, sec. 4.5]).
- No forward reference to a term symbol extends beyond a value definition.

It performs the following transformations:

- Local modules are replaced by variables and classes.[1]
- Calls to case factory methods are replaced by new's.
- Eliminate branches in a conditional if the condition is a constant.

A.3 Simplifying tree rewritings

A.3.1 Uncurry and translate function values to classes

The `uncurry` phase deals with function values and currification. It is discussed in [Gar09b]. The following code snippet illustrates the most important transformations.

```
object Test {
  def curriedFun(x: Int)(y: Int) = x * y
  val fiveTimes = curriedFun(5) _
  val timesFour = curriedFun(_:Int)(4)

  def test() = {
    val dummy1 = curriedFun(7)(3)
    val dummy2 = fiveTimes(2)
    val dummy3 = timesFour(1)
  }
}
```

Before `uncurry`, the internal code is the following:

```
package <empty> {
  final object Test extends java.lang.Object with ScalaObject {
    def this(): object Test = {
      Test.super.this();
      ()
    };
    def curriedFun(x: Int)(y: Int): Int = x.*(y);
    private[this] val fiveTimes: (Int) => Int = {
      ((y: Int) => Test.this.curriedFun(5)(y))
    };
    <stable> <accessor> def fiveTimes: (Int) => Int = Test.this.fiveTimes;
    private[this] val timesFour: (Int) => Int =
        ((x$1: Int) => Test.this.curriedFun((x$1: Int))(4));
```

[1] A module is the internal denomination of an `object`.

```
    <stable> <accessor> def timesFour: (Int) => Int = Test.this.timesFour;
    def test(): Unit = {
      val dummy1: Int = Test.this.curriedFun(7)(3);
      val dummy2: Int = Test.this.fiveTimes.apply(2);
      val dummy3: Int = Test.this.timesFour.apply(1);
      ()
    }
  }
}
```

The uncurry phase transforms it into the following (for brevity, we have omitted the constructors):

```
package <empty> {
  final object Test extends java.lang.Object with ScalaObject {
    def curriedFun(x: Int, y: Int): Int = x.*(y);
    private[this] val fiveTimes: (Int) => Int = {
      {
        @SerialVersionUID(0) final <synthetic> class $anonfun extends
            scala.runtime.AbstractFunction1[Int,Int] with Serializable {
          final def apply(y: Int): Int = Test.this.curriedFun(5, y)
        };
        (new anonymous class $anonfun(): (Int) => Int)
      }
    };
    <stable> <accessor> def fiveTimes(): (Int) => Int = Test.this.fiveTimes;
    private[this] val timesFour: (Int) => Int = {
      @SerialVersionUID(0) final <synthetic> class $anonfun extends
          scala.runtime.AbstractFunction1[Int,Int] with Serializable {
        final def apply(x$1: Int): Int = Test.this.curriedFun((x$1: Int), 4)
      };
      (new anonymous class $anonfun(): (Int) => Int)
    };
    <stable> <accessor> def timesFour(): (Int) => Int = Test.this.timesFour;
    def test(): Unit = {
      val dummy1: Int = Test.this.curriedFun(7, 3);
      val dummy2: Int = Test.this.fiveTimes().apply(2);
      val dummy3: Int = Test.this.timesFour().apply(1);
      ()
    }
  }
}
```

A.3.2 Tail call optimization, Scala only

Although both Scala and Ozma provide tail call optimization, they do it in very different ways. This section describes the Scala flavor of tail call optimization. It is done in the `tailcalls` phase, which is described in [Gar09c].

Taking the example from section 1.2.1 again:

```
object Test {
  def displayList(list: List[Any]) {
    if (!list.isEmpty) {
      println(list.head)
      displayList(list.tail)
    }
  }
}
```

Before tail call optimization, the internal code is:

```
package <empty> {
  final object Test extends java.lang.Object with ScalaObject {
    def this(): object Test = // [...]
    def displayList(list: List[Any]): Unit =
      if (list.isEmpty().unary_!())
        {
          scala.this.Predef.println(list.head());
          Test.this.displayList(list.tail())
        }
      else
        ()
  }
}
```

The `tailcalls` phase rewrites it as:

```
package <empty> {
  final object Test extends java.lang.Object with ScalaObject {
    def this(): object Test = // [...]
    def displayList(list: List[Any]): Unit = {
      <synthetic> val _$this: Test.type = Test.this;
      _displayList(_$this,list){
        if (list.isEmpty().unary_!())
          {
            scala.this.Predef.println(list.head());
            _displayList(Test.this, list.tail())
          }
        else
          ()
      }
    }
  }
}
```

A.3.3 Specialization

The `specialize` phase takes care of specialization. In Scala, generic parameters can be annotated with `@specialize`. This annotation instructs the compiler to generate specialized code for primitive types. Specialization in Scala is discussed in [DO09a]. The following code snippet shows a minimal demonstration of specialization:

```
class Ref[@specialized(Int) A](var value: A) {
  override def toString = value.toString
}

object Test {
  def main(args: Array[String]) = {
    val anyRef = new Ref[Any](6)
    println(anyRef)

    val intRef = new Ref(5)
    println(intRef)
    intRef.value = 2
```

```
    val longRef = new Ref[Long](1)
    println(longRef)
  }
}
```

Before specialization, its internal equivalent is:

```
package <empty> {
  class Ref[@specialized(scala.Int) A >: Nothing <: Any] extends
      java.lang.Object with ScalaObject {
    <paramaccessor> private[this] var value: A = _;
    <accessor> <paramaccessor> def value(): A = Ref.this.value;
    <accessor> <paramaccessor> def value_=(x$1: A): Unit =
      Ref.this.value = x$1;
    def this(value: A): Ref[A] = {
      Ref.super.this();
      ()
    };
    override def toString(): java.lang.String = Ref.this.value().toString()
  };
  final object Test extends java.lang.Object with ScalaObject {
    def this(): object Test = {
      Test.super.this();
      ()
    };
    def test(): Unit = {
      val anyRef: Ref[Any] = new Ref[Any](6);
      scala.this.Predef.println(anyRef);
      val intRef: Ref[Int] = new Ref[Int](5);
      scala.this.Predef.println(intRef);
      intRef.value_=(2);
      val longRef: Ref[Long] = new Ref[Long](1L);
      scala.this.Predef.println(longRef)
    }
  }
}
```

The `specialize` phase transforms it into:

```
package <empty> {
  class Ref[@specialized(scala.Int) A >: Nothing <: Any] extends
      java.lang.Object with ScalaObject {
    <paramaccessor> protected[this] var value: A = _;
    <accessor> <paramaccessor> def value(): A = Ref.this.value;
    <accessor> <paramaccessor> def value_=(x$1: A): Unit =
      Ref.this.value = x$1;
    def this(value: A): Ref[A] = {
      Ref.super.this();
      ()
    };
    override def toString(): java.lang.String = Ref.this.value().toString();
    <paramaccessor> <specialized> def value$mcI$sp(): Int =
      Ref.this.value().asInstanceOf[Int]();
    <paramaccessor> <specialized> def value$mcI$sp_=(x$1: Int): Unit =
      Ref.this.value_=(x$1.asInstanceOf[A]());
    def specInstance$(): Boolean = false
  };
  final object Test extends java.lang.Object with ScalaObject {
    def this(): object Test = {
      Test.super.this();
      ()
    };
    def test(): Unit = {
```

```
      val anyRef: Ref[Any] = new Ref[Any](6);
      scala.this.Predef.println(anyRef);
      val intRef: Ref[Int] = new Ref$mcI$sp(5);
      scala.this.Predef.println(intRef);
      intRef.value$mcI$sp_=(2);
      val longRef: Ref[Long] = new Ref[Long](1L);
      scala.this.Predef.println(longRef)
    }
  };
  <specialized> class Ref$mcI$sp extends Ref[Int] {
    <paramaccessor> <specialized> protected[this] var value$mcI$sp: Int = _;
    <accessor> <specialized> def value$mcI$sp(): Int =
      Ref$mcI$sp.this.value$mcI$sp;
    override <accessor> <specialized> def value(): Int =
      Ref$mcI$sp.this.value$mcI$sp();
    <accessor> <specialized> def value$mcI$sp_=(x$1: Int): Unit =
      Ref$mcI$sp.this.value$mcI$sp = x$1;
    override <accessor> <specialized> def value_=(x$1: Int): Unit =
      Ref$mcI$sp.this.value$mcI$sp_=(x$1);
    <specialized> def this(value$mcI$sp: Int): Ref$mcI$sp = {
      Ref$mcI$sp.super.this(null.asInstanceOf[Int]());
      ()
    };
    def specInstance$(): Boolean = true
  }
}
```

Note that we restrained specialization to apply to the `Int` primitive type. The code grows rapidly when using specialization. In particular, when using several specialized generic types, we get exponential growth. Specialization must then be used with care.

A.3.4 Explicit outer references and pattern matching

The `explicitouter` phase performs two transformations. It rewrites nested classes so that they have an explicit reference to their enclosing classes, and it rewrites `match` statements as conditionals and labels.

Adding explicit outer references is straightforward. Pattern matching in Scala was designed by Burak Emir and is discussed in his PhD. thesis [Emi07].

A.3.5 Erasure

The Java Virtual Machine does not deal with generics. It uses only raw types. Erasure is the transformation of all generic types in a program to raw types. Additionaly, the `erasure` phase makes explicit boxing and unboxing operations. It is discussed in [Gar11a].

We illustrate erasure on this tiny code snippet:

```
object Test {
  def test() {
    val list = List(1, 2, 3)
    list foreach { x => println(x+1) }
  }
}
```

Before erasure, the corresponding internal code is:

```
package <empty> {
  final object Test extends java.lang.Object with ScalaObject {
    def this(): object Test = // [...]
```

```
    def test(): Unit = {
      val list: List[Int] = immutable.this.List.apply[Int](
        scala.this.Predef.wrapIntArray(Array[Int]{1, 2, 3}));
      list.foreach[Unit]({
        @SerialVersionUID(0) final <synthetic> class $anonfun extends
            scala.runtime.AbstractFunction1$mcVI$sp with Serializable {
          def this(): anonymous class $anonfun = // [...]
          final def apply(x: Int): Unit = $anonfun.this.apply$mcVI$sp(x);
          <specialized> def apply$mcVI$sp(v1: Int): Unit =
            scala.this.Predef.println(v1.+(1))
        };
        (new anonymous class $anonfun(): (Int) => Unit)
      })
    }
  }
}
```

After erasure, it looks like this:

```
package <empty> {
  final object Test extends java.lang.Object with ScalaObject {
    def this(): object Test = // [...]
    def test(): Unit = {
      val list: List = immutable.this.List.apply( // note: no [Int]
        scala.this.Predef.wrapIntArray(
          Array[Int]{1, 2, 3})); // note: [Int] still present
      list.foreach({
        @SerialVersionUID(0) final <synthetic> class $anonfun extends
            scala.runtime.AbstractFunction1$mcVI$sp with Serializable {
          def this(): anonymous class $anonfun = // [...]
          final def apply(x: Int): Unit = $anonfun.this.apply$mcVI$sp(x);
          <specialized> def apply$mcVI$sp(v1: Int): Unit =
            scala.this.Predef.println(scala.Int.box(v1.+(1))); // note: boxing
          final <bridge> def apply(v1: java.lang.Object): java.lang.Object = {
            $anonfun.this.apply(scala.Int.unbox(v1));
            scala.runtime.BoxedUnit.UNIT
          }
        };
        (new anonymous class $anonfun(): Function1)
      })
    }
  }
}
```

Note that erasure does not erase generic types of arrays.

A.3.6 Translate lazy values into lazified defs

The `lazyvals` phase takes care of `lazy val`'s. This phase is detailed in [Gar11e].

It converts them to methods with memoization. This phase is not concerned with Ozma lazy execution: `lazy val` and lazy execution are two totally different concepts. A lazy value is evaluated the first time it is *accessed*, not *needed*. Besides, if its execution throws an exception, further attemps to access it will re-evaluate it, until no exception is thrown.

For some reason, part of the job is taken care of by `uncurry`, so that `lazy val` handling is spread on these two phases.

Although less powerful than Ozma lazy execution, we keep this phase intact in Ozma in order to keep the semantics unchanged.

This code snippet illustrates both local and field lazy values:

```
object Test {
  lazy val value1 = {
    println("evaluating value1")
    1
  }

  def main(args: Array[String]) {
    lazy val value2 = {
      println("evaluating value2")
      2
    }

    println("start")
    println(value1)
    println(value2)
    println(value1)
    println(value2)
  }
}
```

Before uncury, the internal code is:

```
package <empty> {
  final object Test extends java.lang.Object with ScalaObject {
    def this(): object Test = {
      Test.super.this();
      ()
    };
    lazy private[this] var value1: Int = {
      scala.this.Predef.println("evaluating value1");
      1
    };
    def main(args: Array[String]): Unit = {
      lazy var value2$lzy: Int = {
        scala.this.Predef.println("evaluating value2");
        2
      };
      scala.this.Predef.println("start");
      scala.this.Predef.println(Test.this.value1);
      scala.this.Predef.println(value2);
      scala.this.Predef.println(Test.this.value1);
      scala.this.Predef.println(value2)
    }
  }
}
```

The uncurry phase first rewrites lazy values as so-called lazy defs:

```
package <empty> {
  final object Test extends java.lang.Object with ScalaObject {
    def this(): object Test = {
      Test.super.this();
      ()
    };
    lazy private[this] var value1: Int = _;
    <stable> <accessor> lazy def value1(): Int = {
      Test.this.value1 = {
        scala.this.Predef.println("evaluating value1");
        1
      };
```

```
      Test.this.value1
    };
    def main(args: Array[java.lang.String]): Unit = {
      lazy var value2$lzy: Int = _;
      <stable> <accessor> lazy def value2(): Int = {
        value2$lzy = {
          scala.this.Predef.println("evaluating value2");
          2
        };
        value2$lzy
      };
      scala.this.Predef.println("start");
      scala.this.Predef.println(scala.Int.box(Test.this.value1()));
      scala.this.Predef.println(scala.Int.box(value2()));
      scala.this.Predef.println(scala.Int.box(Test.this.value1()));
      scala.this.Predef.println(scala.Int.box(value2()))
    }
  }
}
```

The `lazyvals` phase rewrites lazy defs so that they implement memoization, in thread-safe way:

```
package <empty> {
  final object Test extends java.lang.Object with ScalaObject {
    def this(): object Test = {
      Test.super.this();
      ()
    };
    lazy private[this] var value1: Int = _;
    <stable> <accessor> lazy def value1(): Int = {
      Test.this.value1 = {
        scala.this.Predef.println("evaluating value1");
        1
      };
      Test.this.value1
    };
    def main(args: Array[java.lang.String]): Unit = {
      @volatile var bitmap$0: Int = 0;
      lazy var value2$lzy: Int = _;
      <stable> def value2(): Int = {
        if (bitmap$0.&(1).==(0))
          {
            Test.this.synchronized({
              if (bitmap$0.&(1).==(0))
                {
                  value2$lzy = {
                    scala.this.Predef.println("evaluating value2");
                    2
                  };
                  bitmap$0 = bitmap$0.|(1);
                  ()
                };
              scala.runtime.BoxedUnit.UNIT
            });
            ()
          };
        value2$lzy
      };
      scala.this.Predef.println("start");
      scala.this.Predef.println(scala.Int.box(Test.this.value1()));
      scala.this.Predef.println(scala.Int.box(value2()));
```

```
      scala.this.Predef.println(scala.Int.box(Test.this.value1()));
      scala.this.Predef.println(scala.Int.box(value2()))
    }
  }
}
```

A.3.7 Move nested functions and classes to top level

In Scala/Ozma, one can define functions and classes inside methods. Such functions and classes can access (for reading or writing) values and variables that are accessible in the the outer method (these are said to be *free*). Scala makes intensive use of nested classes because of anonymous closures. The `lambdalift` moves nested functions and classes to the closest outer class, while preserving links between free values and variables.

A theoretical approach to the specifications and algorithms of this phase can be found in [Alt06, cpt. 3]. Another approach, as code walkthrough, is given in [Gar11d].

As is explained in section 6.4.2, this phase is specialized in Ozma to support dataflow values.

The following code snippet illustrates some of the rewritings of this phase:

```
object Test {
  def foldL(init: Int, list: List[Int], f: (Int, Int) => Int) = {
    def loop(prev: Int, list: List[Int]): Int =
      if (list.isEmpty) prev else loop(f(prev, list.head), list.tail)
    loop(init, list)
  }

  def main(args: Array[String]) {
    val list = List(1, 3, 4, 8)
    val sum = foldL(0, list, _ + _)
    println(sum)
  }
}
```

Before `lambdalift`, the internal code is the following:

```
package <empty> {
  final object Test extends java.lang.Object with ScalaObject {
    def this(): object Test = {
      Test.super.this();
      ()
    };

    def foldL(init: Int, list: List, f: Function2): Int = {
      def loop(prev: Int, list: List): Int =
        if (list.isEmpty())
          prev
        else
          loop(f.apply$mcIII$sp(prev, scala.Int.unbox(list.head())),
            list.tail().$asInstanceOf[List]());
      loop(init, list)
    };

    def main(args: Array[java.lang.String]): Unit = {
      val list: List = immutable.this.List.apply(
        scala.this.Predef.wrapIntArray(Array[Int]{1, 3, 4, 8}));

      val sum: Int = Test.this.foldL(0, list, {
        @SerialVersionUID(0) final <synthetic> class $anonfun extends
```

```
          scala.runtime.AbstractFunction2$mcIII$sp with Serializable {
        def this(): anonymous class $anonfun = {
          $anonfun.super.this();
          ()
        };

        final def apply(x$1: Int, x$2: Int): Int =
          $anonfun.this.apply$mcIII$sp(x$1, x$2);

        <specialized> def apply$mcIII$sp(v1: Int, v2: Int): Int =
          v1.*(args.length()).+(v2);

        final <bridge> def apply(v1: java.lang.Object,
            v2: java.lang.Object): java.lang.Object =
          scala.Int.box($anonfun.this.apply(
            scala.Int.unbox(v1), scala.Int.unbox(v2)))
      };
      (new anonymous class $anonfun(): Function2)
    });

    scala.this.Predef.println(scala.Int.box(sum))
  }
 }
}
```

After lambda lifting, we get:

```
package <empty> {
  final object Test extends java.lang.Object with ScalaObject {
    def this(): object Test = {
      Test.super.this();
      ()
    };

    def foldL(init: Int, list: List, f$1: Function2): Int =
      Test.this.loop$1(init, list, f$1);

    def main(args: Array[java.lang.String]): Unit = {
      val list: List = immutable.this.List.apply(
        scala.this.Predef.wrapIntArray(Array[Int]{1, 3, 4, 8}));

      val sum: Int = Test.this.foldL(0, list, {
        (new anonymous class $anonfun$1(args$1): Function2)
      });

      scala.this.Predef.println(scala.Int.box(sum))
    };

    /* Notice this unnested method. It has an additional parameter
     * that captures the free variable `f` in the non-lifted version */
    final private[this] def loop$1(prev: Int, list: List,
        f$1: Function2): Int =
      if (list.isEmpty())
        prev
      else
        Test.this.loop$1(f$1.apply$mcIII$sp(prev, scala.Int.unbox(
          list.head())), list.tail().$asInstanceOf[List](), f$1);

    /* Notice this unnested class. It has an additional constructor
     * parameter, as well as a param-accessor field, that capture the
     * free variable `args` in the non-lifted version. */
    @SerialVersionUID(0) final <synthetic> class $anonfun$1 extends
        scala.runtime.AbstractFunction2$mcIII$sp with Serializable {
```

```
      def this(args$1: Array[java.lang.String]): anonymous class $anonfun$1 = {
        $anonfun$1.super.this();
        ()
      };

      final def apply(x$1: Int, x$2: Int): Int =
        $anonfun$1.this.apply$mcIII$sp(x$1, x$2);

      <specialized> def apply$mcIII$sp(v1: Int, v2: Int): Int =
        v1.*($anonfun$1.this.args$1.length()).+(v2);

      final <bridge> def apply(v1: java.lang.Object,
          v2: java.lang.Object): java.lang.Object =
        scala.Int.box($anonfun$1.this.apply(
          scala.Int.unbox(v1), scala.Int.unbox(v2)));

      <synthetic> <paramaccessor> private[this] val
          args$1: Array[java.lang.String] = _
    }
  }
}
```

A.3.8 Write the code of constructors

Until now, most of the code of the primary constructor is still located in the *template*, i.e. the body of the class. The `constructors` phase moves the code to the actual primary constructor. This essentially means initializing all fields (`val`'s and `var`'s) with their respective rhs, and including any statement from the template.

This phase is discussed in [Gar11b].

The following code snippet illustrates the most important role of this phase:

```
class C(val name: String) {
  val msg = "Hello, " + name

  println(msg)
}
```

Before the `constructors` phase, the internal code is:

```
package <empty> {
  class C extends java.lang.Object with ScalaObject {
    <paramaccessor> private[this] val name: java.lang.String = _;
    <stable> <accessor> <paramaccessor> def name(): java.lang.String =
      C.this.name;
    def this(name: java.lang.String): C = {
      C.super.this();
      ()
    };
    private[this] val msg: java.lang.String = "Hello, ".+(C.this.name());
    <stable> <accessor> def msg(): java.lang.String = C.this.msg;
    scala.this.Predef.println(C.this.msg())
  }
}
```

The `constructors` phase fills in the constructor, so that the result is:

```
package <empty> {
  class C extends java.lang.Object with ScalaObject {
    <paramaccessor> private[this] val name: java.lang.String = _;
    <stable> <accessor> <paramaccessor> def name(): java.lang.String =
```

```
      C.this.name;
    private[this] val msg: java.lang.String = _;
    <stable> <accessor> def msg(): java.lang.String = C.this.msg;
    def this(name: java.lang.String): C = {
      C.this.name = name;
      C.super.this();
      C.this.msg = "Hello, ".+(name);
      scala.this.Predef.println(C.this.msg());
      ()
    }
  }
}
```

A.3.9 Eliminate inner classes, Scala only

The JVM has no concept of nested class. All classes must be on the top level (just inside a package). The `flatten` phase moves all nested classes to top level. Ozma does not need this phase because the Ozma engine can accomodate nested classes. This phase is discussed in [Gar11c].

A.3.10 Mixin composition

The `mixin` phase takes care of mixin composition with traits. In Java, interfaces cannot declare fields or give an implementation for their methods. Traits can do both, and it is `mixin`'s responsibility to add to a class extending a trait all the fields and methods it inherits from this trait.

The following code snippet illustrates some effects of the `mixin` phase:

```
trait Trait {
  val name: String
  var field: Int = _

  def print() {
    println(name)
  }

  def incField() {
    field += 1
  }
}

class C(val name: String) extends Trait {
  def decField() {
    field -= 1
  }
}
```

Before mixin composition, the internal code is:

```
package <empty> {
  abstract trait Trait extends java.lang.Object with ScalaObject {
    <stable> <accessor> def name(): java.lang.String;
    <accessor> def field(): Int;
    <accessor> def field_=(x$1: Int): Unit;
    def print(): Unit;
    def incField(): Unit
```

```
  };
  class C extends java.lang.Object with Trait with ScalaObject {
    <paramaccessor> private[this] val name: java.lang.String = _;
    <stable> <accessor> <paramaccessor> def name(): java.lang.String =
      C.this.name;
    def decField(): Unit = C.this.field_=(C.this.field().-(1));
    def this(name: java.lang.String): C = {
      C.this.name = name;
      C.super.this();
      C.this.$asInstanceOf[Trait$class]()./*Trait$class*/$init$();
      ()
    }
  };
  abstract trait Trait$class extends java.lang.Object with
      ScalaObject with Trait {
    private[this] var field: Int = _;
    <accessor> def field(): Int = Trait$class.this.field;
    <accessor> def field_=(x$1: Int): Unit = Trait$class.this.field = x$1;
    def print(): Unit =
      scala.this.Predef.println(Trait$class.this.name());
    def incField(): Unit =
      Trait$class.this.field_=(Trait$class.this.field().+(1));
    def /*Trait$class*/$init$(): Unit = {
      ()
    }
  }
}
```

After mixin, it becomes:

```
package <empty> {
  abstract trait Trait extends java.lang.Object with ScalaObject {
    <stable> <accessor> def name(): java.lang.String;
    <accessor> def field(): Int;
    @scala.runtime.TraitSetter <accessor> def field_=(x$1: Int): Unit;
    def print(): Unit;
    def incField(): Unit
  };
  class C extends java.lang.Object with Trait with ScalaObject {
    <accessor> def field(): Int = C.this.field;
    private[this] var field: Int = _;
    @scala.runtime.TraitSetter <accessor> def field_=(x$1: Int): Unit =
      C.this.field = x$1;
    def print(): Unit = Trait$class.print(C.this);
    def incField(): Unit = Trait$class.incField(C.this);
    <paramaccessor> private[this] val name: java.lang.String = _;
    <stable> <accessor> <paramaccessor> def name(): java.lang.String =
      C.this.name;
    def decField(): Unit = C.this.field_=(C.this.field().-(1));
    def this(name: java.lang.String): C = {
      C.this.name = name;
      C.super.this();
      Trait$class./*Trait$class*/$init$(C.this);
      ()
    }
  };
  abstract trait Trait$class extends  {
    def print($this: Trait): Unit = scala.this.Predef.println($this.name());
    def incField($this: Trait): Unit = $this.field_=($this.field().+(1));
    def /*Trait$class*/$init$($this: Trait): Unit = {
      ()
    }
  }
```

```
}
```

Note how fields and accessors have migrated from the implementation class `Trait$class` to class `C`.

A.3.11 Platform-dependant cleanup, Scala only

The `cleanup` phase was introduced to compile *structural types* for the JVM [DO09b]. It rewrites so-called *dynamic calls* to reflection-based calls. A dynamic call is a call to a method that is defined in a structural type. The `cleanup` phase applies some other transformations, like rewriting `try` blocks in expression-position so that they become statements.

More generally, this phase rewrites statement constructs which the back-end cannot deal with into simpler constructs. This phase is discussed in [Gar09a].

Ozma does not need this phase, because method calls are dynamic in Oz anyway. So we compile dynamic calls as regular calls. The other transformations performed by `cleanup` are also irrelevant for the Mozart back-end.

A.4 The JVM back-end, Scala only

Back-ends are where Scala and Ozma diverge completely. Scala compiles towards the JVM, whereas Ozma compiles towards Mozart. This section describes the back-end phases related to Scala.

A.4.1 Generate icode

The first back-end phase of Scala is `icode`, which generates so-called *icode* from the AST. icode is an intermediate language for stack-based machines. Both the JVM and MSIL are stack-base, which makes icode a great intermediate language for these two back-ends.

icode is structured as a set of `IClass`. Each `IClass` has a set of `IField` and `IMethod`. And `IMethod` contains the code for its body. The code is relatively low-level, consisting of linear instructions, quite like an assembly language. The different instructions of icode are declared in `nsc.backend.icode.Opcodes`. The classes `IClass`, `IField` and `IMethod` can be found in `icode.Members`.

The `icode` phase uses a rather classical code generation algorithm. It is discussed in [Gar10].

A.4.2 Optimizations

As of version 2.9.0, the Scala compiler has three optimization phases on icode, found in the package `nsc.backend.opt`:

- `inliner`: inline some method calls,
- `closelim`: eliminate closures that are, in fact, not called, and
- `dce`: dead code elimination.

A.4.3 Byte-code generation

The last phase of the Scala compiler is `jvm`. It generates `.class` files from icode: one `.class` file for each instance of `IClass`. This includes converting icode to JVM byte-code. The code can be found in `nsc.backend.jvm.GenJVM`.

A.5 The Mozart back-end, Ozma only

In the Ozma compiler, the back-end is completely replaced by a new one, compiling towards Mozart byte-code. It follows the same structure as the JVM back-end, though:

- Generate an intermediate language that is closer to the back-end runtime, here Oz (see section 6.3.2),
- Apply optimizations, here only tail call optimization (see section 6.5), and
- Generate actual byte-code (see section 6.3.3).

Appendix B

Domain Specific Language for digital logic simulation

This appendix describes a DSL for digital logic simulation. It is used in section 4.2.3. The entire source code can be found in `docs/examples/digitallogic/`.

B.1 Bits and signals

First, we define a `Bit`. We do not use simply `Boolean` or `Int` because we want a) type-safety and b) specific operations on these.

```
package digitallogic

import scala.ozma._

sealed abstract class Bit(bool: Boolean) {
  val toBool = bool
  val toInt = if (bool) 1 else 0
  val name = toInt.toString

  override def toString() = name

  def unary_~() =
    if (this eq One) Zero else One

  def | (right: Bit) =
    if ((this eq One) || (right eq One)) One else Zero

  def & (right: Bit) =
    if ((this eq One) && (right eq One)) One else Zero

  def ^ (right: Bit) =
    if (this eq right) Zero else One

  def ~& (right: Bit) = ~(this & right)
  def ~| (right: Bit) = ~(this | right)
  def ~^ (right: Bit) = ~(this ^ right)
```

```
}

case object Zero extends Bit(false)
case object One extends Bit(true)
```

Using these definitions, we can use straightforward operations on bits, e.g. `One & Zero` yields `Zero`. All possible operations on one or two bits are available as *operators*. This is nice.

A *signal* is modeled as a stream of bits, i.e. `List[Bit]`. For convenience and documentation purpose, we define a shortcut for this one:

```
object Signal {
  type Signal = List[Bit]
}

package object digitallogic {
  type Signal = Signal.Signal
}
```

B.2 Simple gates

Signals are composed using *gates*. Consider the simplest gate, which negates a signal. It is a function working on a signal, hence a stream of bits, and returning another signal. This is straightforward to implement using `map`:

```
def not(input: Signal): Signal =
  intput map (~_)
```

However, this would not work as is in a logic *circuit*, because all gates must execute simultaneously. Hence, we need to wrap this computation in a thread (and use `toAgent` to achieve proper memory management):

```
def not(input: Signal): Signal =
  thread(input.toAgent map (~_))
```

Now how do we define two-input gates, e.g. `and`? Here we have to use `zip` and `map` from collections. In a `toAgent` setting, there is a simplified `zipMap`:

```
def and(left: Signal, right: Signal) =
  thread(left.toAgent.zipMap(right)(_ & _))
```

As there are several two-input gates, we have to copy-and-paste this several times. We do not want this, so we factorize the common structure:

```
private def makeGate(combination: (Bit, Bit) => Bit)(
    left: Signal, right: Signal): Signal = {
  thread(left.toAgent.zipMap(right)(combination))
}

def and(left: Signal, right: Signal) = makeGate(_ & _)(left, right)
```

We can even simplify the definition of `and` using currification:

```
val and  = makeGate(_ & _) _
```

Now we can trivially implement all other two-input gates:

```
val or   = makeGate(_ | _) _
val nand = makeGate(_ ~& _) _
val nor  = makeGate(_ ~| _) _
val xor  = makeGate(_ ^ _) _
def xnor = makeGate(_ ~^ _) _
```

There is also the `delay` gate:

```
def delay(input: Signal): Signal = Zero :: input
```

B.3 Operators for gates

With the gate definitions of the previous section, we have to write `and(x, y)` to obtain the signal $x \cdot y$. We would like an infix operator view. We cannot do it like we did for `Bit` because `Signal` is only a shortcut for `List[Bit]`, not a dedicated class. We solve this problem using a *view*.

A view is an implicit conversion from one type to another. We define it using the `implicit` keyword:

```
package object digitallogic {
  type Signal = Signal.Signal

  implicit def signal2ops(signal: Signal) = new SignalOps(signal)
}
```

We define the operations we want to be able to perform on signals in the class `SignalOps`:

```
class SignalOps(signal: Signal) {
  import Gates._

  def unary_!() = not(signal)

  def && (right: Signal) = and(signal, right)
  def || (right: Signal) = or(signal, right)
  def !&& (right: Signal) = nand(signal, right)
  def !|| (right: Signal) = nor(signal, right)
  def ^^ (right: Signal) = xor(signal, right)
  def !^^ (right: Signal) = xnor(signal, right)
}
```

Now we can perform operations on signals using infix operators:

```
val z = x && y
```

The compiler will translate this to:

```
val z = signal2ops(x).&&(y)
```

Note that because the constructor of `SignalOps` does not wait for `x`, and because the `and` function executes in a thread, this statement does not block if given unbound values (either for `x` or `y`). This is an important property that is necessary for this DSL to be successful.

B.4 Building signals from scratch

Now we can easily compose existing signals. The last contribution to the DSL is to build elementary signals from scratch. The `Signal` object provides these means:

```
object Signal {
  type Signal = List[Bit]

  def apply(bits: Bit*): Signal = List(bits:_*)
  def unapplySeq(signal: Signal): Some[List[Bit]] = Some(signal)

  def clock(delay: Int = 1000, value: Bit = One) = {
    def loop(): Signal = {
      sleep(delay)
      value :: loop()
    }

    thread(loop())
  }

  def generator(clock: Signal)(value: Int => Bit) = {
    def loop(clock: Signal, i: Int): Signal = {
      waitBound(clock)
      value(i) :: loop(clock.tail, i+1)
    }

    thread(loop(clock, 0))
  }

  def cycle(clock: Signal, values: Bit*) = {
    generator(clock) {
      i => values(i % values.length)
    }
  }
}
```

The `apply` method allows for creating finite signals like in:

```
val x = Signal(1, 0, 0, 1, 0)
```

The `unapplySeq` is not used in the examples. It allows to use signals in pattern matching constructs, like `List`. We provide it for completeness, though there is probably no practical use case for pattern matching in the context of digital logic simulation.