Compiling Scala for the Java Virtual Machine

Michel Schinz

École Polytechnique Fédérale de Lausanne

Laboratoire des méthodes de programmation
Institut d'Informatique Fondamentale
Faculté Informatique et Communication
École Polytechnique Fédérale de Lausanne

Contents

List of Figures

List of Tables

Version abrégée

Scala est un nouveau langage de programmation, développé à l'EPFL, combinant les principales caractéristiques des langages orientés-objets et fonctionnels. Scala a été conçu pour interagir aisément aussi bien avec Java qu'avec .NET. Pour ce faire, son compilateur peut produire des programmes s'exécutant sur la machine virtuelle de chacune de ces deux plateformes.

Cette thèse se focalise sur la compilation de deux concepts importants de Scala : l'héritage par *mixins* et les types à l'exécution. Les techniques de compilation sont présentées dans le cadre de la machine virtuelle Java, mais pourraient être adaptées sans difficulté à d'autres environnements similaires.

L'héritage par *mixins* est une forme d'héritage relativement récente, plus puissante que l'héritage simple, mais évitant la complexité de l'héritage multiple. Nous proposons deux techniques de compilation de cet héritage : par copie de code, et par délégation. La mise en œuvre actuelle, basée sur la copie de code, est ensuite présentée puis justifiée.

Par types à l'exécution on entend la possibilité qu'a un programme d'examiner, lors de son exécution, le type de ses valeurs. Cette possibilité est d'une part intéressante en soit, puisqu'elle offre de nouveaux moyens au programmeur. D'autre part, elle est requise pour la mise en œuvre d'autres concepts de haut niveau, comme le filtrage de motifs, la sérialisation sûre, ou la réflection. Nous proposons une technique de compilation basée sur la représentation des types sous forme de valeurs, et montrons comment il est possible d'utiliser les types à l'exécution de la machine virtuelle sous-jacente comme base pour les types à l'exécution de Scala.

Les techniques présentées dans cette thèse ont été mises en œuvre dans le cadre de notre compilateur Scala, nommé `scalac`. Cela nous a permis d'évaluer la qualité des techniques proposées, en particulier leur impact sur les performances des programmes. Cette évaluation s'est faite sur des programmes réels, de taille importante.

Abstract

Scala is a new programming language developed at EPFL and incorporating the most important concepts of object-oriented and functional languages. It aims at integrating well with the Java and .NET platforms: their respective libraries are accessible without glue code, and the compiler can produce programs for both virtual machines.

This thesis focuses on the compilation of two important concepts of Scala : *mixin inheritance* and *run time types*. The compilation techniques are presented in the context of the Java virtual machine, but could be adapted easily to other similar environments.

Mixin inheritance is a relatively recent form of inheritance, offering new capabilities compared to single inheritance, without the complexity of multiple inheritance. We propose two implementation techniques for mixin inheritance: code copying and delegation. The implementation used in the Scala compiler is then presented and justified.

Run time types make it possible for a program to examine the type of its values during execution. This possibility is interesting in itself, offering new capabilities to the programmer. Furthermore, run time types are also required to implement other high level concepts, like pattern matching, type-safe serialisation and reflection. We propose an implementation technique based on the representation of types as values, and show how to use the run time types of the underlying virtual machine to implement those of Scala.

The techniques presented in this thesis have been implemented in our Scala compiler, `scalac`. This enabled us to evaluate these techniques, in particular their impact on the performances of programs. This evaluation was performed on several real, non-trivial programs.

Acknowledgements

I would first like to thank my adviser Prof. Martin Odersky for accepting me in his group and for his support during the time I spent at LAMP.

My colleague Philippe Altherr, who shared both his office and his many insights about programming languages, helped me immensely during all those years. His efforts to clean up the internals of the Scala compiler saved me countless hours of debugging, and generally made it a pleasure to work on the project.

Vincent Cremet spent a lot of time discussing various issues with me, and proof-read this thesis very carefully, providing many valuable comments about it. His help was very much appreciated.

Many people contributed to the Scala project — and its predecessor Funnel — and working with all of them proved very enjoyable. Besides the persons already mentioned, I would therefore like to thank Iulian Dragos, Gilles Dubochet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Christine Röckl, Erik Stenman, Christoph Zenger and Matthias Zenger.

The second half of the LAMP, composed of Sébastien Briais, Daniel Bünzli, Johannes Borgström, Rachele Fuzzati, Simon Kramer, and Prof. Uwe Nestmann provided great company at lunch, and many interesting subjects for discussion during our coffee breaks.

I would also like to express my gratitude to the members of my jury, Prof. Rachid Guerraoui, Prof. Paolo Ienne, Prof. Shriram Krishnamurthi and Prof. Mirko Viroli, for taking the time to read my thesis.

Finally, the moral support of my family and Sabine was invaluable, especially during the final months.

Chapter 1

Introduction

1.1 Scope

Scala is a new programming language developed by Martin Odersky and his team at LAMP [19]. Scala aims at being a practical language, usable for large-scale programs, and incorporating several advanced concepts from recent research.

From the beginning, Scala was designed to run on the Java Virtual Machine (JVM) and on the .NET Common Language Runtime (CLR). Even though the design of the language was often directed to ease the integration with these platforms, the compilation of Scala still presented several challenges.

This thesis explains and evaluates the solutions we developed for the implementation of two important concepts of Scala : mixins and run time types. It focuses on the Java platform, because run time types are currently available only on that platform.

All the techniques described here have been implemented in the Scala compiler, `scalac`. This compiler is officially distributed since January 2004, and used in several courses at EPFL. It has also been used to develop a few middle-sized projects, including several compilers — the front-end of current Scala compiler (`scalac`), the forthcoming new Scala compiler (`nsc`), and a compiler for the Scaletta calculus [3] — as well as Web applications — an auction site [18], and the bug-tracker used for the Scala project. These projects have given us the opportunity to evaluate our techniques on realistic programs.

1.2 Background

This section presents the background necessary to understand the rest of the thesis. We start with a quick overview of the Scala language itself, followed by a description of the Java virtual machine and finally the overall architecture of the current Scala compiler.

1.2.1 The Scala programming language

Scala attempts to blend smoothly concepts taken from object-oriented and functional programming languages.

From the object-oriented world, it inherits the concept of classes, which plays a central role in that all values manipulated are viewed as being instances of a class. Classes can be built incrementally using two inheritance mechanisms: single inheritance, and a form of mixin inheritance.

From functional languages, Scala borrows algebraic data-types, pattern matching, closures and anonymous functions. These concepts are all integrated coherently, in a way that fits with the object-oriented nature of the language: algebraic data-types are a special kind of classes called *case classes*, and closures are objects with a single method.

Scala is a typed programming language, featuring an advanced type system. This type system is described in detail in the official language specification [20], which we will not reproduce here. It is however interesting to quickly review its major features, which are also presented using examples in appendix A:

Polymorphism Classes and methods can take type parameters, and these parameters can have both lower and upper bounds, which can refer to the parameter itself (F-bounded polymorphism).

Type members A class can include type members as well as value members; like value members, type members can be either abstract or concrete. Abstract type members — also known as virtual types [31] — are bounded by a type, and this bound can be refined by subclasses.

Compound types A value can be declared to have a compound type, consisting of a set of component types, which must be class types, and a refinement. The refinement, possibly empty, constrains the type of members of the components, by giving them more precise types. A compound type is close to an intersection type [25], as it expresses the fact that a value has simultaneously all the types of the components.

Singleton types A basic form of dependent types is provided by singleton types, which refer to a value. There are always two values of a given singleton type: **`null`** and the value to which the type refers.

Scala also tries to integrate well with current popular platforms: both Java and .NET classes are viewed as if they were normal Scala classes, from which one can even inherit. This automatically makes a large body of libraries available to the programmer.

The core aspects of Scala have been formalised in the *νObj* calculus [21].

1.2.2 The Java virtual machine

The Java virtual machine [16] (JVM) is a stack-based machine targeted towards the execution of compiled Java programs. It receives as input a single *class file*, containing the definition of the main class of the program, and executes a particular method of that class. As the execution proceeds, the class files corresponding to the other classes needed by the program are loaded on demand.

One of the novel aspects of the JVM is that it verifies the class files after loading them, to make sure that their execution will not violate some defined safety properties. The checks performed by the verifier include a complete type checking of the program [13].

To make this type checking possible, all the instructions of the JVM are typed: the type of the operands they expect on the stack as well as the type of the result they push back can be inferred from the instructions themselves or their parameters. The type language of the JVM is very close to the one of the Java language itself prior to version 5, *i.e.* without generics: a type is a class or interface name, or one of the nine primitive types (**`boolean`**, **`int`**, etc.).

Although very Java centric, the JVM has been used extensively over the recent years as a target platform for many compilers. A web page listing such compilers and interpreters currently contains close to 190 entries [32]. This popularity seems to be due mostly to two factors:

1. a large body of Java libraries is available, and targeting the JVM instantly gives access to them,

2. the JVM is available on all major platforms, and compiled Java programs run on all of them without needing any recompilation.

However, since the JVM is tailored to Java, compiling languages which differ widely from Java can be challenging.

1.2.3 The Scala compiler

The Scala compiler, `scalac`, transforms a Scala program into either a set of Java class files or an MSIL assembly file, depending on the chosen target. Currently, run time types are only supported on the Java platform, and for that reason we will ignore the .NET platform in the rest of this document.

It should however be noted that version 2.0 of the .NET common language runtime (CLR) will provide full support for generics, including run time types [12]. This could be used to implement run time types for Scala on that platform, provided that a good mapping from Scala types to CLR types can be devised. This is no easy task, though, as the type system of Scala contains many features which do not appear in the one of .NET, *e.g.* variance annotations, singleton types and compound types.

`Scalac` is organised as a set of successive phases, which can broadly be split into two groups:

1. the front end phases, which perform the analysis of the input file to make sure that it is a valid Scala program, and produce a fully attributed abstract syntax tree (AST),

2. the back end phases, which gradually simplify the AST, to the point where it is very similar to Java or C# code; at that point, a byte code generation phase produces the actual class or assembly files which constitute the final output of the compiler.

Each phase is composed of two transformation functions: a tree transformer, and a type transformer. The tree transformer is applied to the AST of the program being compiled, while the type transformer is applied to the type of all identifiers needed during compilation—be they defined in the AST or loaded from files compiled separately.

Figure 1.1 on the facing page illustrates the organisation of the compiler.[1] Phases in bold will be described in detail in the rest of this document. Each phase is listed in the order in which it is applied, along with a short description of the concept it translates. Most phases work by transforming an input tree (or type) in which their concept can appear, into one where this concept has been translated away. For example, before the UnCurry phase, curried functions can appear in the tree or in the types of imported identifiers, but after it, they will all have been transformed into uncurried, multi-argument functions.

[1] One minor optimisation phase, which transforms directly recursive tail calls, was left out.

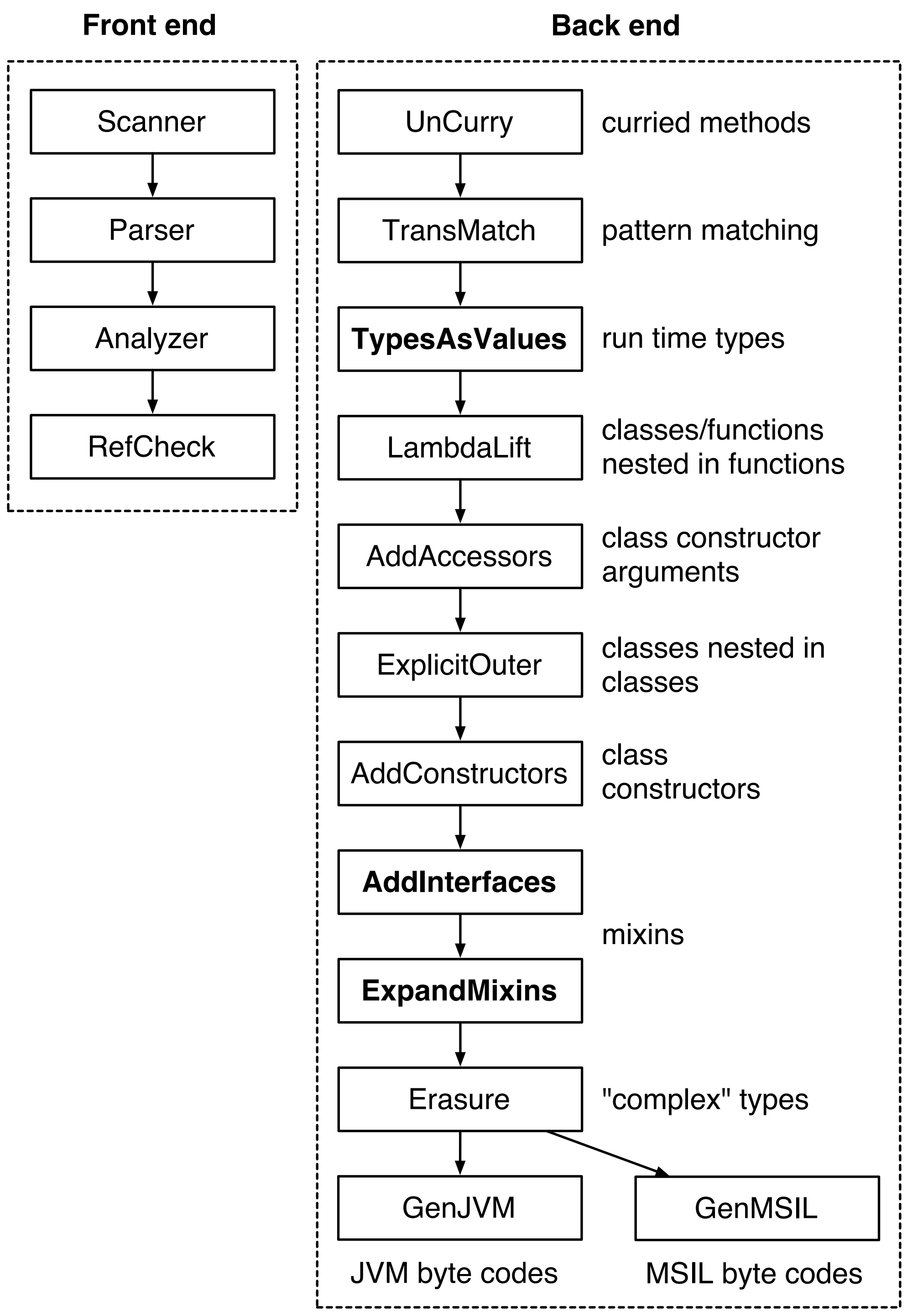


Figure 1.1: Phases of the Scala compiler

Some phases which are not the main subject of this thesis still play an important role in its understanding. These phases are: TransMatch, LambdaLift, ExplicitOuter, AddConstructors and Erasure. They deserve being explained in some details, and are therefore the subject of the following sections.

The TransMatch phase

TransMatch translates pattern-matching expressions (see [20, chapter 7]) into conditional expressions that make use of tags attached to values as well as run time types to match values with patterns. This makes TransMatch the biggest user of run time types, as we will see in chapter 3. Pattern matching is a central concept of Scala, and the importance of run time types is mostly due to the fact that they are key to its implementation.

The LambdaLift phase

Scala classes and functions can be defined inside many contexts: classes, functions, blocks, etc. Additionally, classes can be defined directly inside of a package, and are then said to be at the top level. Top-level classes which do not appear in the scope of an explicit package declaration are implicitly placed inside of a special, anonymous package.

The aim of the LambdaLift phase is to lift classes and functions so that the former appear either at the top level or directly inside other classes, and the latter are (possibly private) methods of classes. During this lifting, free variables appearing in lifted entities — classes or functions — get added to them as parameters. When such variables are mutable, they are turned into reference cells.

The ExplicitOuter phase

In Scala, all nested classes have full access to the members of their enclosing class(es). The task of ExplicitOuter is to lift nested classes to the top level by augmenting them with an explicit reference to their enclosing object. This reference is then used to access all the members of enclosing classes.

All phases coming after ExplicitOuter therefore see a program composed only of top-level classes.

The AddConstructors phase

In Scala, a class does not contain an explicit method definition for its constructor, like it does for example in Java. Rather, the class itself can take arguments, which are visible in its whole body, including any nested classes. Furthermore, the body of a class can contain arbitrary expressions, which are evaluated each time a new instance is created.

This implicit constructor is called the *primary constructor* of the class. It is also possible to define secondary constructors as members of the class named **`this`**, but all have to end with a call the primary constructor.

The aim of AddConstructors is to make the primary constructor explicit by turning it into a method called `<init>`. This method takes the same arguments as the class. Its body is composed of all the expressions which originally appeared in the body of the class itself, and of calls to constructors of the super-class and mixins, if any.

After AddConstructors, the class does not have value parameters anymore, and its body is composed exclusively of member definitions, one of which is the constructor `<init>`. At this stage, and with respect to constructors, the class is very similar to a standard Java or C# class.

The Erasure phase

Erasure performs a partial erasure of the types appearing in a program. The aim of this partial erasure is to transform all the Scala types appearing in the program into JVM or CLR types. The exact transformation performed in Scala will be described in chapter 4.

1.3 Inheritance graph notation

Several inheritance graphs appear in the remainder of this document. These graphs are composed of classes and interfaces, related by standard inheritance, mixin inheritance or interface implementation. The notation is presented in figure 1.2.

Interfaces do not exist in Scala, but the AddInterfaces phase creates interfaces as part of its translation. It is therefore useful to have a notation for them.

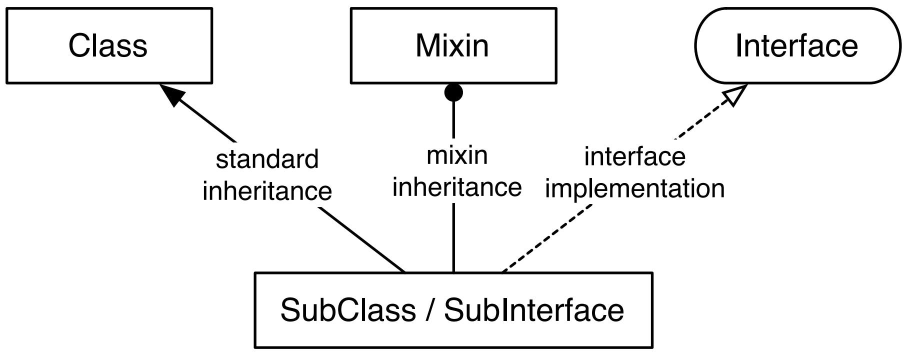


Figure 1.2: Inheritance graph notation

Chapter 2

Mixins

2.1 Introduction

To support code reuse, all object-oriented languages support some notion of inheritance. The simplest and most common form is single inheritance which, while powerful and adequate in many cases, is usually perceived as being too limited to solve all reuse problems. To overcome these limitations, several new forms of inheritance have been proposed: multiple inheritance, mixin inheritance, and recently trait inheritance.

Scala offers a kind of mixin inheritance which is slightly different from what exists in other mixin-based languages. For that reason, before detailing its implementation, we present it after reviewing the various kinds of inheritance mentioned above.

2.1.1 Single inheritance

Single inheritance is the most common, and simplest, form of inheritance. It enables a class to inherit all the members (methods and fields) from another class, called its superclass, and possibly override some of them with new definitions.

2.1.2 Multiple inheritance

Multiple inheritance is a generalisation of single inheritance by which a class can inherit simultaneously from several parent classes. Some limitations of single inheritance are thereby removed, but additional problems are introduced.

A well-known one is the *diamond problem*, where a class inherits a single ancestor twice, from two different parents. While inheriting methods several times is not a problem, as it cannot be observed, the repeated inheritance of mutable fields certainly is. Apart from wasting space in the heir class, it raises tricky questions about which copy of a field is modified or read when.

Multiple inheritance also has limitations which make certain kinds of abstractions difficult to express. For example, it doesn't provide a way to abstract over an extension of many classes with the same set of members. This kind of abstractions is precisely what mixins provide, as detailed below.

2.1.3 Mixin inheritance

Mixins are partial class definitions, whose superclass is abstract [7]. When applied to a class, they yield a new subclass of that class, obtained by adding or overriding members. Basically, mixins can be seen as (meta) functions which, when given a class, produce a new subclass of that class.

Two mixins can be composed together to form a composite mixin. Applying that composite to a class means applying its second component to the class, resulting in a new class to which the first component of the composite is applied to obtain the final class. In other words, mixins are composed like functions.

Composite mixins open the door to a variant of the diamond problem: when two composite mixins are composed, it is possible that both composites have a component in common. Applying the resulting mixin to a class results in the same set of members to be added twice.

Apart from that problem, the fact that mixins have to be composed in a given order has been criticised by Schärli *et al.* [29]. They argue that an order which is appropriate for all the methods of the mixins to compose does not always exist. To solve that problem, they propose *trait inheritance*, an unordered variant of mixin inheritance, which is examined next.

2.1.4 Trait inheritance

Traits are, like mixins, partial class definitions which can be incorporated into classes [29]. They are slightly more restricted than mixins in that they cannot contain mutable fields. This restriction avoids the problem of diamond inheritance, since repeated inheritance of methods is not a problem, as we have seen.

Trait inheritance is similar to mixin inheritance, the key difference being the lack of order among traits: when a set of traits is inherited by a class, they are all inherited simultaneously, and any member defined by more than one trait results in an ambiguity. This ambiguity must be explicitly resolved by providing an overriding definition in the class.

The lack of order among traits also has a consequence on the meaning of the **super** keyword. When it appears in a trait, it always refers to the superclass of the class inheriting the trait. This differs from mixins, where **super** refers to the previous mixin in the composition. This behaviour extends naturally to the occurrences of **super** appearing in the body of the class itself: with mixin inheritance, they refer to the mixin inherited last, whereas with traits they always refer to the superclass.

We will see in §2.5.5 that along with trait inheritance comes a special form of **super**, providing access to any method of any inherited trait. No equivalent usually exists with mixin inheritance.

2.2 Mixin inheritance in Scala

Apart from single inheritance, Scala supports a form of mixin inheritance which does not correspond exactly to either mixin or trait inheritance as described above. Despite its name, it is actually closer to the latter than to the former, as we will now see.

A class `C` can be declared to inherit from one superclass `S` and several mixins `M1`, `M2`, . . ., `Mn`. This is written as follows:

```
class C extends S with M1 with M2 with ... with Mn { ... }
```

For such a definition to be valid, the type of the superclass `S` must be a subtype of the type of the superclass of every mixin `M1` to `Mn`. It is easy to understand the motivation of this restriction: since `S` will be used as the actual superclass of all the mixins, its type should guarantee that it can be.

Like for trait inheritance, the order of the various mixins does not matter in Scala, and any concrete member defined in more than one of them must be overridden in the heir class to resolve the ambiguity.

Both single and mixin inheritance are tied with subtyping. This means that with the above definition, the type `C` is a subtype of the type of classes `S` and `M1` to `Mn`.

Mixins do not exist as a separate concept in Scala. Rather, any (non-final) class can be inherited either through standard single inheritance, or through mixin inheritance. The way it is inherited determines which members get inherited by its heir: with single inheritance, all members are inherited, while with mixin inheritance only the ones introduced or overridden by the mixin class itself or its own mixins are. In other words, when a class is used as a mixin, only the difference between it and its superclass is considered.

To avoid the diamond problem mentioned before, it is illegal in Scala to inherit a single mixin through more than one path, unless the mixin in question is declared as being a trait. A trait is a special kind of abstract class, restricted so that it can be repeatedly inherited without problem. The restrictions on traits are:

1. The primary constructor of a trait cannot have value parameters, and secondary constructors are disallowed altogether — this ensures that when a trait is inherited multiple times, all copies receive the same (empty) constructor arguments;

2. The construction of a trait cannot have side effects — this ensures that multiple inheritance of a trait will not lead to multiple evaluation of side effects, and potentially to distinguishable copies of the same trait;

3. The parent constructors of a trait cannot have value parameters — this ensures that all copies of a trait inherited repeatedly pass the same arguments to their superclasses (*i.e.* none),

4. A trait only inherits from other traits, not standard mixins, through mixin inheritance — this ensures that the rules just presented cannot be circumvented by hiding mixins behind traits.

2.2.1 Mixin example

To illustrate the translation steps performed by the Scala compiler, we use a small example. We start by defining a class representing an integer counter, with one method to obtain the current value of the counter, and one to increment it:

```
class Counter {
  protected var v: Int = 0;
  def value: Int = v;
  def increment: Unit = (v = v + 1);
}
```

Then, we define several subclasses of that class, each providing a different variant of the counter. All these subclasses are supposed to be mixed together to obtain the desired composite behaviour.

The first of these subclasses provides a method to reset the counter:

```
trait ResettableCounter extends Counter {
  def reset: Unit = (v = 0);
}
```

The second one provides a method to bring the counter back to some earlier state:

```
abstract class UndoableCounter extends Counter {
  private var history: List[Int] = List();

  override def v_=(newV: Int): Unit = {
    history = (v :: history);
    super.v_=(newV);
  }
  def undo: Unit = {
    super.v_=(history.head);
    history = history.tail
  }
}
```

Notice the redefinition of v_=, which is the setter method for the field v, implicitly introduced by the compiler when v was defined[1].

We can now define a class for a resettable and undoable counter by composing the appropriate mixins:

```
class RUCounter
  extends Counter
  with ResettableCounter
  with UndoableCounter;
```

2.3 Encoding mixins on the JVM

The first step in implementing Scala mixin inheritance is to find a way to encode it on the JVM, which does not offer it as such.

There are two aspects to the encoding of mixin inheritance on the JVM: first, since the language of the JVM is typed, we must make sure that our encoding produces well-typed JVM programs; second, we must find a

[1]In Scala, all fields are accessed and updated via getter and setter methods automatically introduced by the compiler. The programmer can override these methods in subclasses, like here.

way to share the code of the mixins among all classes which inherit from them.

2.3.1 Typing aspect

For the typing aspect, we need to ensure that the type of a class which inherits from superclass `S` and mixins `M1` to `Mn` is a subtype of all of them. This is a form of multiple subtyping, which suggests the use of interfaces to express the subtyping aspect of mixin inheritance on the JVM: our encoding must provide one interface per mixin, and any class inheriting from a set of mixins must implement the corresponding interfaces.

2.3.2 Code sharing aspect

To share the code of the mixins among their heir classes, we can either copy it, or really share it at the level of the JVM using delegation. We explore these two options below.

Sharing by copying

Like for all forms of inheritance, the code sharing aspect of mixin inheritance can be implemented by simple code copying: each time a class inherits from a mixin, the code from the latter is copied into the former. This approach is relatively easy to implement, and produces efficient code. Its main disadvantage is its cost in terms of code size.

Sharing by delegation

To try to avoid the cost of code copying, it is possible to share code using delegation [15]. The idea is that classes hold one delegate object per mixin from which they inherit, and forward calls to mixin methods to these delegates.

With this approach, calls to methods of the superclass appearing in mixins are problematic: they cannot be resolved statically, as they usually are, because the exact method to call depends on the context in which the mixin is used. This is illustrated by figure 2.1, which shows two mixins `M1` and `M2`, both redefining method `m` of class `A`. It is clear that when `M1` is inherited by `B`, the call **`super`**`.m` it contains refers to `A`'s version of `m`; but when the same `M1` is inherited by `E`, the call **`super`**`.m` refers to `D`'s version of `m`, itself inherited from `M2`.

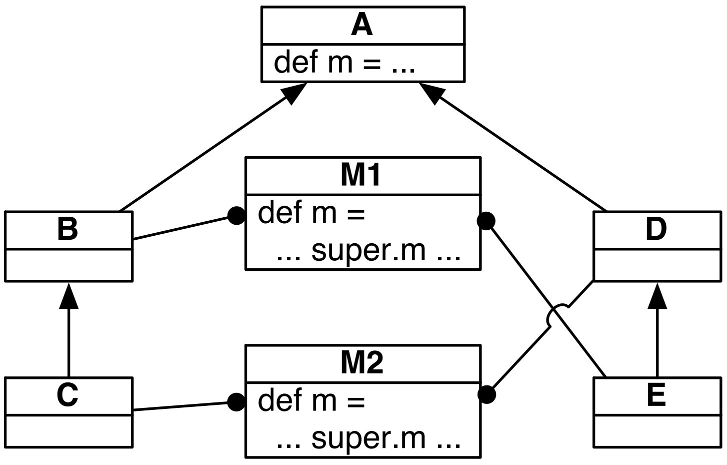


Figure 2.1: Super calls and mixins

To solve this problem, delegate objects should have a way to identify the location of their heir class in the hierarchy. This can be obtained by associating a level to every class, which is the length of the path leading from that class to the top of the hierarchy, following only single-inheritance links. In our example, class `A` could be at level `0`, classes `B` and `D` at level `1`, and classes `C` and `E` at level `2`.

With this in place, it is relatively easy to provide access to any version of an overridden method, using *super-accessor* methods. Every method of every class has such a method associated to it. The super-accessor gets the same arguments as the original method, as well as the level of the class which should perform the super call.

For example, the super-accessor method `super_m` below provides access to all versions of method `m` for classes `A`, `B` and `C`:

```
class A {
    void m() { /* ... */ }
    void super_m(int lvl) { throw new Error(); }
}
class B extends A {
    void m() { /* forward to delegate for M1 */ }
    void super_m(int lvl)
        { if (lvl == 1) super.m(); else super.super_m(lvl); }
}
class C extends B {
    void m() { /* forward to delegate for M2 */ }
    void super_m(int lvl)
        { if (lvl == 2) super.m(); else super.super_m(lvl); }
}
```

Given an instance of class `C`, calling `super_m` with an argument of 1 calls `m` in `A`, while an argument of 2 calls `m` in `B`.

These super-accessor methods can be used by mixins to perform super calls. On creation every delegate object receives the level of the class inheriting from it. To perform a super call, it invokes the super-accessor method of its delegator, passing it that level.

In our example, mixin `M1` would look as follows:

```
class M1Delegate {
    private final int lvl;
    public M1Delegate(int lvl) { this.lvl = lvl; }
    void m(A self) {
        /* ... */ self.super_m(lvl); /* ... */
    }
}
```

All copies of `M1Delegate` used by instances of `B` would get a level of 1 on creation, while the ones used by instances of `E` would get a level of 2. This ensures that super calls performed by `M1Delegate` are routed appropriately.

Example translations

To illustrate the two translation techniques for mixins just presented, we show in figure 2.2 how class `RUCounter` could be written in Java using code copying, and in figure 2.3 how it could be written using delegation.

Notice that in the second translation we have made the assumption that mixins are not normal classes, since we have translated them differently. To shorten the code, we also provided only one implementation of a super-accessor method, `super_setV`, while in reality such methods should be provided for all methods of all classes.

2.3.3 Encoding used by scalac

To translate mixins, `scalac` uses a combination of multiple interface inheritance and code duplication: JVM interfaces are used to provide the proper subtyping relationships, and code copying is performed to import code from mixins into heir classes.

```
interface ResettableCounter { public void reset(); }
interface UndoableCounter { public void undo(); }

class Counter {
    private int v = 0;

    public int getV() { return v; }
    public void setV(int newV) { v = newV; }

    public int value() { return getV(); }
    public void increment() { setV(getV() + 1); }
}

class RUCounter extends Counter
  implements ResettableCounter, UndoableCounter {
    public void reset() { setV(0); }

    private LinkedList history = new LinkedList();

    public void setV(int newV){
        history.addFirst(new Integer(getV()));
        super.setV(newV);
    }
    public void undo() {
        super.setV(((Integer)history.removeFirst()).intValue());
    }
}
```

Figure 2.2: Mixins in Java implemented through code copying

```
interface ResettableCounter { public void reset(); }
class ResettableCounterDelegate {
    public static void reset(Counter self) { self.setV(0); }
}
interface UndoableCounter { public void undo(); }
class UndoableCounterDelegate {
    private LinkedList history = new LinkedList();
    private final int level;
    public UndoableCounterDelegate(int l) { level = l; }
    public void setV(Counter self, int newV){
        history.addFirst(new Integer(self.getV()));
        self.super_setV(level, newV);
    }
    public void undo(Counter self) {
        int last = ((Integer)history.removeFirst()).intValue();
        self.super_setV(level, last);
    }
}
class Counter {
    private int v = 0;

    public int getV() { return v; }
    public void setV(int newV) { v = newV; }
    public void super_setV(int level, int newV) { throw new Error(); }
    public int value() { return getV(); }
    public void increment() { setV(getV() + 1); }
}
class RUCounter extends Counter
    implements ResettableCounter, UndoableCounter {
    UndoableCounterDelegate ucDelegate =
        new UndoableCounterDelegate(1);
    public void reset() { ResettableCounterDelegate.reset(this); }
    public void undo() { ucDelegate.undo(this); }
    public void setV(int newV) { ucDelegate.setV(this, newV); }
    public void super_setV(int level, int newV) {
        if (level == 1) super.setV(newV);
        else super.super_setV(level, newV);
    }
}
```

Figure 2.3: Mixins in Java implemented through delegation

This translation is performed by two successive phases: AddInterfaces introduces interfaces and then ExpandMixins copies code. These two phases are described in detail in the rest of this chapter.

Code duplication was chosen instead of delegation for performance reasons. Currently, any Scala class can be used either as a mixin or as a normal class. If delegation was used to implement mixins, all the code of methods would have to be put in delegates. Therefore, *every* Scala method call would result in two JVM method calls: one to the forwarding method, and one to the delegate method.

That said, even if Scala distinguished explicitly mixins from normal classes, it is not evident that a delegation-based approach would be interesting. The overhead of super-accessor and forwarding methods would first have to be measured on real programs as it might be important, especially in terms of performance.

2.4 Interfaces for types

As we have seen, the concepts of types and classes are usually tied in object-oriented languages, and Scala and Java are no exceptions. However, Java provides the notion of interface which defines a type without defining a class at the same time.

This makes it possible to completely untie inheritance and subtyping, by strictly adhering to the following discipline: always use interfaces to represent types, and inheritance solely to reuse code. This implies ignoring completely the types automatically created for classes, and always explicitly specifying the type(s) of a class using interfaces.

Such an approach is not really practical for a Java programmer, as it implies a lot of redundant declarations. But for a compiler targeting the JVM, it is very interesting as it provides a lot of freedom in building the subtyping relation independently of the inheritance relation. This freedom is precisely what is needed to implement mixin inheritance in `scalac`. The first step of the mixin translation is therefore to make sure that the program distinguishes types from classes.

2.4.1 The AddInterfaces phase

AddInterfaces splits all class definitions in a program into two parts: an interface to represent the type, and a class to represent the class.

The interface is given the name of the original class, while the class gets a `$class` suffix added to its name.

The interface contains a declaration for all the methods of the class, including private ones which are renamed first, as we will see in §2.4.4.

The subtyping hierarchy of the original program is reflected by the interfaces, in that the interface for a given class is made to implement all the interfaces of the class' parents, mixins included. If the class originally implemented a Java interface, this interface is also implemented by the class' interface. We will see below that a small problem appears with classes inheriting from Java classes, as no interface exists for them.

Once the subtyping hierarchy has been constructed using interfaces, the link between classes and their type is made by simply adding the appropriate interface to the parent of each class.

Finally, the program is transformed so that interfaces are used for types, and classes for themselves. This implies replacing references to the original class with the new interface, except in two locations: instance creation expressions (**`new`**) and the parents of a class still refer to the class.

This process is illustrated in figures 2.4 and 2.5 on the next page, which show the inheritance hierarchy for the `RUCounter` class and its parents — `Object` excepted — before and after AddInterfaces[2].

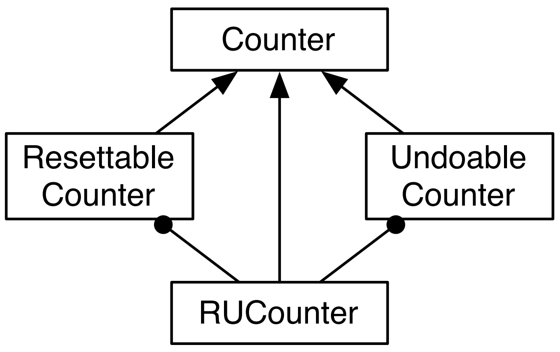


Figure 2.4: Counter class hierarchy before mixin expansion

2.4.2 Unmixable classes

Almost all classes in the original program receive an interface, but there are exceptions. Since the aim of these interfaces is to enable mixin inheritance, only classes which can actually be used as mixins need one. The following kinds of classes are not usable as mixins, and therefore do not get an interface:

[2]In these figures, the `$class` prefix has been shortened to `$c` to save space.

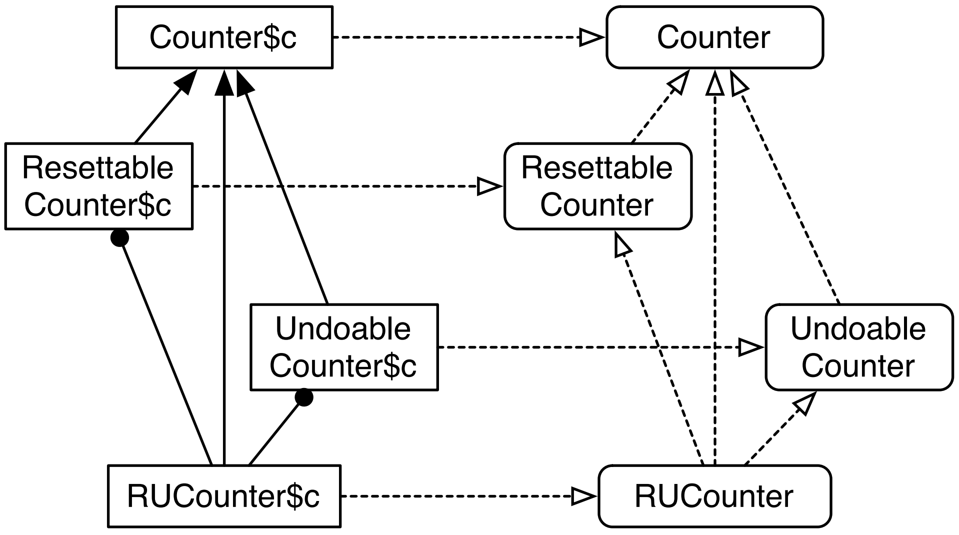


Figure 2.5: Counter class hierarchy after AddInterfaces

1. classes corresponding to Scala singleton objects [20, §5.4], since they are not exposed to the programmer, and

2. classes created by the compiler to represent closures, since they are not visible as classes in the original program.

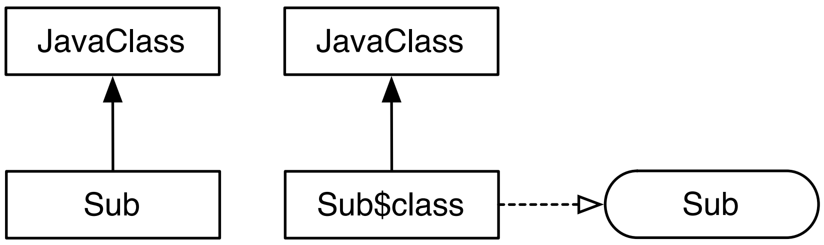
Strictly speaking, only classes which actually *are* used as mixins should get an interface. But in the presence of separate compilation, it is not possible to determine whether a class will be used as a mixin or not. AddInterfaces therefore makes the conservative assumption that all classes which can be used as mixins will be.

To obtain a better approximation of the classes in need of an interface, an analysis of the whole program is required. Alternatively, mixins could be explicitly distinguished from normal classes in Scala.

2.4.3 Subclasses of Java classes

The translation just presented is problematic for Scala classes which inherit from a Java class other than `Object`. Figure 2.6 on the following page illustrates the translation by AddInterfaces of such a class, called `Sub` and inheriting from Java class `JavaClass`. After AddInterfaces, all instances of `Sub$Class` are manipulated as values of type `Sub`, and unfortunately the fact that these instances also have type `JavaClass` has been lost in the translation.

To work around this problem, the Scala compiler inserts a type cast each time a value of type `Sub` is used as a value of type `JavaClass`. These casts are safe, in the sense that they will never fail at run time. This is due to the fact that the only values of type `Sub` are either instances of `Sub$class` or one of its subclasses, or instances of a class `C` which inherits from `Sub` as a mixin. In the latter case, `C` will have to be a subclass of `JavaClass` too, as required by Scala's notion of valid mixin inheritance.



(a) before AddInterfaces

(b) after AddInterfaces

Figure 2.6: Translation of a Scala class inheriting from a Java class

2.4.4 Private members

Ideally, private members should not be part of the interface created by AddInterfaces. But in some cases it is actually not possible to exclude them, as the following example illustrates:

```
class C {
  private def privateFun: Int = 123;
  def f(c: C): Int = c.privateFun;
}
```

After AddInterfaces, the argument `c` passed to `f` will have the type of the *interface* `C`, and it is clear that if `privateFun` is not part of that interface, the underlined call is not legal.

A similar problem appears with private members which are accessed from nested classes.

To avoid these problems, private members are exposed in the interface, after having been renamed uniquely.

2.5 Inlining mixin code

Once mixin interfaces have been created, mixin code can be copied in the heir classes, which is the task of the ExpandMixins phase.

2.5.1 The ExpandMixins phase

The ExpandMixins phase traverses the mixins of a heir class, and copies all their code into that class. Once this is done, mixin inheritance has been translated away, and the mixins can be removed from the parents of the heir class. This will not affect the subtyping relation, as it has been preserved using the interfaces introduced by AddInterfaces.

The process is illustrated by figure 2.7 which shows the hierarchy of figure 2.5 after its transformation by ExpandMixins. The only difference between this figure and the previous one is that the two mixin inheritance arrows have disappeared.

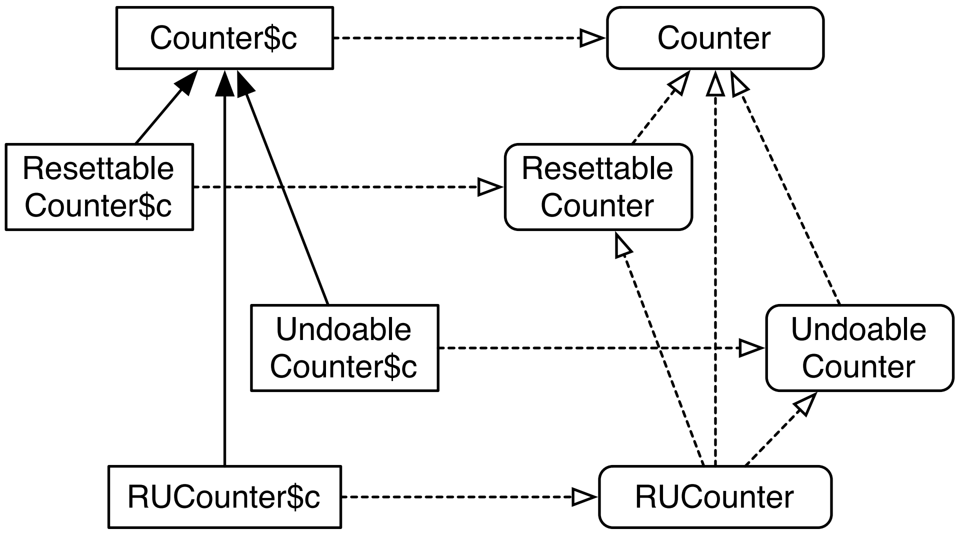


Figure 2.7: Counter class hierarchy after ExpandMixins

Notice that since a mixin can itself inherit from other mixins, it is important to start by expanding mixin bodies before importing them in the heir class.

2.5.2 Mixin type parameters

When importing the code of a mixin, its type parameters, if any, must be replaced by their actual value, given by the heir class. As an illustration, consider the following mixin which, given a method to produce a random element, defines a method to obtain a random list:

```
trait RandomList[T] {
  def random: T;
  def randomList(n: Int): List[T] =
    if (n == 0) List() else random :: randomList(n-1);
}
```

When used in a heir class, the type parameter `T` must be replaced by its actual value. For example, the following object makes use of that mixin to produce random numbers or lists thereof:

```
object RandomDouble with RandomList[Double] {
  def random: Double = Math.random();
}
```

After mixin expansion, all occurrences of `T` have been replaced by `Double` in object `RandomDouble`:

```
interface RandomList[T] {
  def random: T;
  def randomList(n: Int): List[T];
}

object RandomDouble with RandomList[Double] {
  def randomList(n: Int): List[Double] =
    if (n == 0) List() else random :: randomList(n-1);
  def random: Double = Math.random();
}
```

2.5.3 Mixin constructors

The AddConstructors phase comes just before AddInterfaces (figure 1.1), and this means that the two phases concerned with mixin expansion see classes with explicit constructors.

This makes constructors easy to handle during mixin expansion, as they behave like standard methods, with two exceptions:

1. the call to the constructor of the superclass has to be removed, since

it is superseded by the one of the heir class,

2. because all constructors have the same name (`<init>`), the ones inherited from mixins must be renamed uniquely to avoid collision with the constructor of the heir class.

A future redesign of Scala will probably syntactically distinguish mixins from normal classes, and mixins will not contain calls to the constructor of the superclass. This would remove the need to delete these calls during mixin expansion.

2.5.4 Private members

Private members of mixins cannot be simply copied into the heir class, as their name might clash with the name of some other imported member. For that reason, they are given a unique name as they are imported.

2.5.5 Overridden members

Mixin members that are overridden in the heir class can generally be omitted during copying, unless they are accessed through *mixin super reference* [20, §6.3]. For example, in the code sample below, the `print` method of class `C` calls the method `print` of mixin `M` even though it overrides it:

```
class M { def print: Unit = Console.println("M") }

class C with M {
  override def print: Unit = {
    super[M].print;           // prints M
    Console.println("C");     // prints C
  }
}
```

It is clear that the `print` method of mixin `M` has to be imported too, but under a new and unique name. It also has to be made private in the process. The mixin super reference can then be transformed to refer to that imported member, which is now accessed through **`this`** instead of **`super`**.

2.5.6 Nested classes

As illustrated in figure 1.1, mixin expansion happens after phase ExplicitOuter. This means that the program seen by AddInterfaces and Expand-

Mixins does not contain nested classes anymore. As a consequence, classes nested inside of mixins are not copied during mixin expansion.

Notice that it would actually not be possible to perform mixin expansion before phase ExplicitOuter, as classes would then not be closed over their enclosing class. This is illustrated by the following example:

```
class Outer {
  var x: Int = 1;
  class Inner {
    def incX: Unit = { x = x + 1; }
  }
}
object Main {
  val o = new Outer;
  class C with o.Inner;
}
```

If class `Inner` was simply inlined into class `C` without first being lifted to the top level, variable `x` would become free in `C`.

2.6 Example translation

To illustrate the translation of mixins, figure 2.8 shows the translation of the declaration of class `RUCounter`. Not surprisingly, the translated code is very close to its Java equivalent presented in figure 2.2.

2.7 Cost of code copying

To get a rough idea of the cost of code copying, we measured the amount of duplicated code due to mixins in the standard Scala library and in several benchmark programs. This was done by comparing the size of the class files with mixin expansion disabled and enabled. When mixin expansion is disabled, the code produced is not valid and cannot be executed, but we are only interested in measuring its size.

The Scala standard library gets compiled to a total of 1.53 MB of class files. By disabling mixin expansion, this number drops by 10%, to 1.39 MB. Table 2.1 presents the five packages of the library which experience the most important growth because of mixin expansion. For each package, the overall code growth is reported along with the maximal growth for a single file. Notice that even though single files can grow in enormous

```
interface RUCounter extends Counter
  with ResettableCounter with UndoableCounter;

class RUCounter$class extends Counter$class with RUCounter {
  // part imported from ResettableCounter
  private def mixin$ResettableCounter$class$init$: Unit = ();

  def reset: Unit = v_=(0);

  // part imported from UndoableCounter
  private def mixin$UndoableCounter$class$init$: Unit = {
    mixin$UndoableCounter$class$history$ = Nil;
  };

  private var mixin$UndoableCounter$class$history$: List[Int] =
    _;
  def UndoableCounter$history: List[Int] =
    mixin$UndoableCounter$class$history$;
  def UndoableCounter$history_=(x$0: List[Int]): Unit =
    mixin$UndoableCounter$class$history$ = x$0;

  override def v_=(newV: Int): Unit = {
    UndoableCounter$history_=(v :: UndoableCounter$history);
    super.v_=(newV)
  };

  def undo: Unit = {
    super.v_=(UndoableCounter$history.head);
    UndoableCounter$history_=(UndoableCounter$history.tail)
  };

  // part corresponding to RUCounter itself
  def <init>: Unit = {
    super.<init>;
    mixin$ResettableCounter$class$init$;
    mixin$UndoableCounter$class$init$;
  }
}
```

Figure 2.8: Translation of `RUCounter` class

proportions, the overall growth is modest for all packages.

The relatively important growth of Scala's collection library (second and third rows) is not a surprise, as this library makes heavy use of mixins to reuse code. The growth of `scala.xml.parsing` is slightly pathological, being due to a single, big mixin.

	Code growth	
Package	**maximum**	**overall**
`scala.xml.parsing`	1'020%	38%
`scala.collection.immutable`	288%	23%
`scala.collection.mutable`	335%	21%
`scala.xml.dtd`	170%	10%
`scala.testing`	80%	6%

Table 2.1: Top code growth in the Scala library

We additionally measured the growth for four real Scala programs: the current Scala compiler (`scalac`), the new Scala compiler (`nsc`), a partial compiler for the Scaletta calculus (`scaletta`), and an object file printer for Scala (`scalap`). All these benchmarks are described in more detail in section 6.2.1. As can be seen in table 2.2, the overall growth observed for these programs is even lower than for the Scala library. Once again, even though the growth of single files is sometimes impressive, *e.g.* in the `nsc` case, this does not carry on to the total application size.

	Code growth	
Program	**maximum**	**overall**
`scalac`	11%	<1%
`nsc`	2'160%	4%
`scaletta`	187%	1%
`scalap`	11%	1%

Table 2.2: Code growth for Scala programs

Chapter 3

Run Time Types

3.1 Introduction

Types are, by definition, static entities: they get attributed to terms by the type-checker during compilation, and are then usually thrown away. Some languages do keep them, however, and make them available to the programmer at run time, as values. Such languages are said to provide *run time types*.

Run time types play an important role in the implementation of several features of modern programming languages, like pattern matching, reflection or serialisation. Apart from these high-level features, the programmer is usually given indirect access to run time types through two basic operations:

1. the membership test, which tests whether a value is of a given type, and
2. the dynamic type cast, which gives a more precise type to a value than the one inferred by the compiler, deferring the type test until run time.

This chapter and the next three will examine the features related to run time types available in Scala, then describe and evaluate their implementation. We start by explaining in more details the high-level concepts which need run time types, then move on to Scala.

3.1.1 Pattern matching

Pattern matching is a construct which makes it possible to compare a value against a set of patterns, and execute different code depending on the first pattern matching the value. Typically, that code is executed in a context where various parts of the matched value are bound to variables appearing in the matching pattern.

For some languages, given a type, it is possible to know at compile time the set of possible variants of that type. This is for example true of all languages which provide non-extensible algebraic datatypes, like SML. For such languages, full run time types are not needed to implement pattern matching: simple tags are enough. We will see below that Scala is not such a language, however, since its pattern matching construct is powerful enough to require full run time types in general.

3.1.2 Reflection

Reflection refers to the ability of a program to manipulate (parts of) itself at run time, by reifying the abstractions of the programming languages it was written in.

To implement reflection, some information that is typically discarded during compilation must be kept until run time. The exact nature of this information depends on the capabilities offered by reflection, but might include the name of the methods provided by an object, their visibility, and so on. The type of values is part of this information, and therefore run time types are required to implement reflection for a typed language.

3.1.3 Type-safe serialisation

Serialisation (also called marshaling or pickling) is the process of converting a graph of objects into a stream of bytes, to be stored on disk or transferred across a network.

If objects are serialised without also recording their type, deserialisation becomes an unsafe operation, as it is possible to deserialise a value with a different type than the one it had during serialisation. This can result in arbitrary behaviour of the program. This is a well-known problem of Objective Caml's serialisation API, for example: deserialising some datum with an incorrect type can crash the system [14].

To provide type-safe deserialisation, types must therefore be serialised along with values, and this is of course only possible if types exist at run time.

3.2 Run time types in Scala

Scala has a very rich language of types, described in detail in its reference (see [20, chap. 3]). Implementing run time types amounts to compiling this language, first by providing a representation for types as values, and then by translating type expressions to corresponding value expressions working on these type representations. A way to access and manipulate these type representations at run time must also exist for run time types to be of interest.

3.2.1 The language of types

The type terms of Scala are either values, described below, or type expressions which reduce to type values.

Scala offers a relatively standard set of type expressions: polymorphic classes and methods introduce abstraction, which is eliminated by type application. These expressions are translated to corresponding value expressions by the TypesAsValues phase, described in chapter 5.

The type values of Scala are composed of the following:

Singleton types which have the form x.type where x is a value.

Class types which have the form $P\#C[T_1, \ldots, T_n]$ where P is the *prefix*, C is a class name and $T_1, \ldots, T_n$ are the type parameters. Top-level classes have an empty prefix, while nested classes have either a singleton type or another class type as prefix. When P is a singleton type of the form p.type, value p is called the *outer instance* and the type $p.\text{type}\#C[T_1, \ldots, T_n]$ is usually written as $p.C[T_1, \ldots, T_n]$. The type parameters are empty if class C is monomorphic.

Compound types which have the form T_1 with T_2 ... with $T_n\{R\}$ where $T_1 \ldots T_n$ are the component types and R is a refinement.

At run time, these types are represented by instances of Java classes, which will be described in chapter 4.

3.2.2 Run time types of values

All values in Scala are objects — that is, instances of some class — including numbers, characters, booleans and the unit value. The run time type of a value is defined as being the class type corresponding to the class of which this value is an instance.

All Scala values being also JVM values, they have in fact two run time types: the Scala one, and the JVM one. As we will often need to talk about these run time types below, we introduce two operators to denote, respectively, the Scala and JVM run time type of a value: $\texttt{typeof}_S$ and $\texttt{typeof}_J$.

3.2.3 Operations on run time types

In Scala, the two basic operations provided on run time types are the membership test and the type cast, modelled after Java. They are provided by the following two methods, defined in the `scala.Any` class and therefore available for all values:

```
def isInstanceOf[T]: Boolean;  // membership test
def asInstanceOf[T]: T;        // dynamic type cast
```

The meaning of these methods is the following:

- v`.isInstanceOf[`T`]` has type `Boolean` and evaluates to true if and only if value v is not null and has type T at run time,
- v`.asInstanceOf[`T`]` has type T and evaluates to v if value v has type T at run time, and fails with a `ClassCastException` otherwise.

The meaning of `isInstanceOf` can be expressed using the $\texttt{typeof}_S$ operator introduced before:

$$v.\texttt{isInstanceOf}[T] \Leftrightarrow \texttt{typeof}_S(v) <:_S T \tag{3.1}$$

where $<:_S$ is Scala's subtyping relation, defined in [20, §3.5.2] where it is called *conformance*.

The `asInstanceOf` method is also used to perform conversion among numeric types. Therefore, if v is a number and T is a numeric type, then v`.asInstanceOf[`T`]` never throws an exception, but returns the number v converted to a number of type T, with a possible loss of information. A consequence of this special behaviour for numeric types is that it is possible for `asInstanceOf` to succeed even if `isInstanceOf` returns false.

3.2.4 Pattern matching

Most uses of run time types in Scala are the result of the translation of pattern matching. There are two kinds of patterns which need run time types: constructor patterns and typed patterns (see [20, chap. 7]). We examine them below and show how they get translated, by the TransMatch phase, to calls to the two primitives presented above.

Constructor patterns

Constructor patterns check whether a value is an instance of a given case class, and if it is, optionally bind fresh variables to the components of the value.

To illustrate their translation, we use the following small program. It defines classes to represent lists, and later introduces a function `length` to compute the length of such a list.

```
abstract class List[+T];
case class Cons[+T](hd: T, tl: List[T]) extends List[T];
case object Nil extends List[All];

object Main {
  def length[T](l: List[T]): Int = l match {
    case Cons(_, tl) => 1 + length(tl)
    case Nil => 0
  }
}
```

The `length` method uses a constructor pattern for the `Cons` case, and is translated as follows:

```
def length[T](l: List[T]): Int = {
  var $result: Int = _;
  if (if (l.isInstanceOf[Cons[T]]) {
        val tl: List[T] = l.asInstanceOf[Cons[T]].tl();
        $result = 1 + length(tl);
        true
      } else if (l == Nil) {
        $result = 0;
        true
      } else
       false)
    $result
  else
     scala.MatchError.report(l)
}
```

When more than two patterns appear in a `match` expression, the compiler uses a tag attached to objects to dispatch to the appropriate case, instead of a chain of conditionals like here. A membership test nevertheless has to be performed in each of the cases, unless the class of the value being matched is *sealed* [20].

We will see in sections 5.7 and 5.8 that it is sometimes possible to optimise this code, to avoid paying the full cost of the membership test and type cast.

Typed patterns

Typed patterns test whether a value is of a given type *T* and, if it is, bind the value to a fresh variable of type *T*.

To illustrate the translation of these patterns, consider the implementation of the method `equals` for a Scala class representing integer cells:

```
class IntCell(val x: Int) {
  override def equals(thatAny: Any): Boolean = thatAny match {
    case that: IntCell => x == that.x
    case _ => false
  }
}
```

The TransMatch phase translates this code into the following, which makes use of `isInstanceOf` and `asInstanceOf`, and closely resembles what one would write in Java :

```
class IntCell(val x: Int) {
  override def equals(thatAny: Any): Boolean = {
    if (thatAny.isInstanceOf[IntCell]) {
      val that: IntCell = thatAny.asInstanceOf[IntCell];
      x == that.x
    } else
      false
  }
}
```

3.2.5 Implementation restrictions

Before presenting our implementation of run time types for Scala, we review its limitations. They will be justified later, and possible techniques to overcome them will be proposed for future work. These limitations are:

1. Only prefixes composed of a singleton type are supported (see §5.11).

2. Compound types with non-empty refinements are only partially supported: it is for example not possible to perform a membership test with a refined compound type as argument (see §4.3.7).

3. Array types do not support all operations: the creation of array types with singleton or compound types is prohibited, and the membership test and type cast fail in some well-defined circumstances (see §4.3.5).

Because of these limitations, it is possible that some operations on run time types cannot be performed correctly. However, we have designed our implementation in such a way that these cases never go unnoticed: all operations on run time types either fail with an exception, or produce a correct answer.

Chapter 4

Representing Run Time Types

4.1 Introduction

The implementation of run time types is composed of two parts: first, a library to represent Scala types as JVM values, offering the operations necessary to manipulate them at run time; and second, a set of transformations performed at compile time to encode the types as values. This chapter will detail the first part, and the next one will concentrate on the second part.

Before looking at the representation of Scala types as JVM objects, we examine how Scala types are mapped to JVM types, since the latter has an important influence on the former.

4.2 Mapping source types to JVM types

Like all compilers targeting the JVM, the Scala compiler must make sure that it produces programs which are type-correct, in order to be accepted by the JVM byte-code verifier. This implies that it must map source (Scala) types to target (JVM) types.

Since the JVM provides its own run time type operations — the INSTANCEOF and CHECKCAST instructions [16] — it is natural to ask whether these can be used to implement run time types for Scala, too. To answer that question, we first look at the possible mappings of Scala types to JVM types, then present the one used by the Scala compiler and examine how it simplifies the implementation of run time types for Scala.

4.2.1 Full erasure

At one extreme, it is possible to map *all* source types to a single target type, which could be `Object` for the JVM, but any arbitrary empty interface would also do. We will say that this mapping performs the *full erasure* of source types, because all information contained in the types is lost. Obviously, such a mapping doesn't provide any help when implementing run time types.

Full erasure is generally not used by real compilers for typed languages, unless they produce untyped code.

4.2.2 Isomorphic mapping

At the other extreme, the mapping of types can be an isomorphism: each different source type is mapped to a different target type, in such a way that the subtyping relation is preserved. Such a mapping is interesting because run time type operations on source types can be directly translated to the equivalent operations on target types.

An example of such a mapping is the trivial one performed by Java compilers before generics were added to the language: source and target types were then basically the same. A more interesting example of an isomorphic mapping is NEXTGEN's, which encodes type parameters into the name of JVM types. It is explored in more detail in section 4.2.5.

4.2.3 Partial erasure

Between these two extremes lie a range of mappings which we will call *partial erasures*, which preserve some parts of the source types and erase the rest. The idea behind these mappings is to make source types as simple as needed for them to correspond to JVM types, but not simpler.

Since part of the source types is preserved by these mappings, run time types for the source language can generally be implemented partly using the run time types of the JVM. But to provide precise run time types, the erased parts of source types have to be preserved until run time, and this is accomplished by representing them as values.

A well-known example of a partial erasure mapping is the one used for Java 5 [11, §4.6]. In that case, the erasure consists of removing type parameters and replacing type variables by the erasure of their bounds. Since the erased parts of the source types are *not* preserved until run time, operations on run time types are only provided for a subset of the types, called *reifiable types* (see [11, §4.7]). These are the types for which the partial erasure is the identity.

There are other partial erasure mappings in use, and we will shortly examine below the one proposed by Viroli and Natali [36]. As we will see, the Scala compiler itself uses a partial erasure to map Scala types to JVM types.

4.2.4 Erasure of terms

It should be noted that the transformation of types during compilation cannot be performed alone: it must be accompanied by a transformation on terms, and the two must work hand-in-hand to ensure that all well-typed source programs are turned into well-typed target programs. For example, all the erasure schemes mentioned above require the code to be augmented with type casts, which aim at restoring the typing information lost by the erasure.

4.2.5 Mapping examples

Before describing the mapping we chose for Scala, we examine two mappings used by other language implementations targeting the JVM. These implementations are:

1. the NEXTGEN compiler, which compiles an extension of Java with generic types, and uses an isomorphic mapping,
2. the LM translator, which compiles another extension of Java with generic types, and performs partial erasure of types, preserving the erased parts using Java's reflection capabilities.

The NEXTGEN mapping

NEXTGEN [8] extends Java with the ability to define parametric classes and methods, with a syntax similar to the one of Java 5. For example, a class to represent pairs of values is defined as follows:

```
class Pair<T1,T2> {
  T1 fst; T2 snd;
  public Pair(T1 fst, T2 snd) {
    this.fst = fst; this.snd = snd;
  }
}
```

The main difference between NextGen and Java 5 is that the former provides full support for run time types, through the mapping presented below.

The NextGen mapping of types is isomorphic, and all source types are represented as JVM interfaces. Since JVM types are simple names, instantiations of polymorphic types are mapped to interfaces with a mangled name, which includes the name of all type parameters.

For example, the NextGen type `Pair<Object,String>` is mapped to the Java interface `Pair$Object$String$`[1]. Since it is in general not possible to know at compile time which instantiations will be performed at run time, these interfaces cannot be generated before run time. Their creation is therefore performed on demand, by a special class loader which is part of the NextGen run time system.

The NextGen mapping presents the advantage of being very light, and of delegating most of the hard work to the JVM. However, like all isomorphic mappings, it makes the very strong assumption that all source types can be represented as JVM types, which are simple names. While completely realistic in the context of NextGen, this assumption cannot be satisfied easily for Scala, as we will see in section 4.2.6.

The LM mapping

The LM translator of Viroli and Natali [36] translates an extension of Java with parametric classes and methods to standard Java. The language accepted by LM is very similar to Java 5, but operations on run time types are not restricted.

LM maps types through a partial erasure similar to the one performed by Java 5. At run time, the erased parts of the types are represented as JVM objects, using Java's reflection API. Operations on run time types are performed with the help of that API.

4.2.6 The Scala mapping

Having examined some existing mappings, we now turn to the one used by the Scala compiler. We start by presenting it without justification, and then explain the rationale for our choices.

[1] In reality, the mapping is more complicated, as it includes the fully qualified name of type parameters, but this is irrelevant for what follows.

Overview

The Scala compiler uses a partial erasure mapping of types, denoted $|\cdot|$ and defined as follows [20, §3.6]:

- The erasure of a type variable is the erasure of its upper bound.
- The erasure of a parameterized type $T[T_1 \ldots T_n]$ is $|T|$.
- The erasure of a singleton type x.type is the erasure of the type of x.
- The erasure of a type projection $T\#x$ is $|T|\#x$.
- The erasure of a compound type T_1 `with ... with` T_n $\{R\}$ is $|T_1|$.
- The erasure of an array type `Array[`T`]` is $|T|$`[]`,
- The erasure of types `All`, `AllRef`, `Any` and `AnyRef` is `Object`,
- The erasure of every other type is the type itself.

This function erases nested types to types containing projections. For example, the erasure of the type corresponding to a class `Inner` nested inside a class `Outer` is `Outer#Inner`. As this is not a valid JVM type, name mangling is used in the same way as in Java to obtain a valid type name, in this case `Outer$Inner`.

The erased parts of the types are represented as Java objects, which are described in detail in section 4.3.

Rationale

The Scala mapping was chosen because Scala's type system is too rich to be mapped fully to the JVM type system, à la NEXTGEN. This is due to the following properties:

1. Scala's type parameters can be contravariant,
2. Scala's type parameters can be instantiated by singleton types which depend on *values*,
3. the type of a nested Scala class depends on the identity of the *instance* into which it is nested (this can be seen as a special case of instantiation with singleton types).

Only the first characteristic really prevents us from completely mapping Scala types into JVM types, but the other two would pose important implementation problems, detailed later.

Contravariance of type parameters prevents a mapping like the one used by NEXTGEN because it requires extending the subtyping relation at run time in ways which are not permitted by the JVM. For example, consider a class with a contravariant type parameter defined as follows:

```
class OutputStream[-T];
```

Now imagine that, at run time, the following classes are loaded, in the given order:

1. `Object`,
2. `OutputStream[Object]` (mapped to `OutputStream$Object`),
3. `String`,
4. `OutputStream[String]` (mapped to `OutputStream$String`).

Because of the contravariance of `OutputStream`'s type argument, and because `String` is a subtype of `Object`, `OutputStream[Object]` must be a subtype of `OutputStream[String]`. Unfortunately, the JVM only permits the addition of *subtypes* to existing types at run time, not super-types. This is a problem here, since `OutputStream[String]` would need to be added as a super-type of `OutputStream[Object]` at point 4, after the latter is loaded.

The fact that Scala type parameters can be instantiated to singleton types, which depend on values, poses another problem: a requirement of an isomorphic mapping like NEXTGEN's is that all types can easily be mapped to strings, and singleton types do not satisfy this requirement. The problem is that to map a singleton type as a string, it should be possible to map *any value* to a string. While not technically infeasible, this implies at least the management of a single global table associating string representation to objects, which would consume both time and memory.

Finally, nested classes pose a problem very similar to the one posed by singleton types. In fact, the type of the outer instance can be seen as a type parameter passed to the inner class. This type parameter is always instantiated with a singleton type representing the outer instance. The problem of how to represent this singleton type as a string reappears.

Properties

Having presented the Scala mapping, we now examine its properties. As explained before, we are mainly interested in knowing whether, and how, the run time types provided by the JVM can help with the implementation of run time types for Scala.

First, it is interesting to note that the erasure used by the Scala compiler does *not* preserve subtyping, because of the way it erases compound types. This is easily seen with a counterexample. Here are two Scala types which are in a subtyping relation — actually they are even equivalent:

$$C \texttt{ with } D <:_{\mathrm{S}} D \texttt{ with } C$$

However, this is not true of their erasure, as we can see:

$$|C \texttt{ with } D| = C \not<:_{\mathrm{J}} D = |D \texttt{ with } C|$$

This distinguishes the Scala erasure from other erasures which preserve subtyping, like the one used in FGJ (see [10, Lemma A.3.5]). The type system of FGJ is much simpler, however, and in particular doesn't include compound types.

Despite this, we can still use the run time types of the JVM in interesting ways to implement Scala's run time types. Indeed, we will see that the following implication holds:

$$v.\texttt{isInstanceOf}[T] \Rightarrow \mathcal{J}(v) \textbf{ instanceof } |T| \tag{4.1}$$

where $\mathcal{J}(v)$ represents the JVM value corresponding to the Scala value v.

This property is interesting because it makes it possible to implement the membership test of Scala in two stages: In the first stage, the test is performed on the erased type — a fast operation; if this test fails, the failure of the whole test can be immediately reported. Otherwise, more elaborate and costly tests have to be performed on the full Scala type.

We can get an intuition of why this property is true by first rewriting it as follows, using the definitions of `isInstanceOf` and **`instanceof`** :

$$\begin{gathered} v.\texttt{isInstanceOf}[T] \Rightarrow \mathcal{J}(v) \textbf{ instanceof } |T| \\ \texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} T \Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} |T| \end{gathered} \tag{4.2}$$

To convince ourselves that this last implication is true, we can do a case analysis on the type T. There are seven cases to consider.

Case 1 The first case is when T is the type of a Java class C.

$$\begin{array}{lr}
\texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} T & \\
\Rightarrow \texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} C & \text{(value of } T\text{)} \\
\Rightarrow v \text{ is instance of a subclass of } C & \text{(property of Scala)} \\
\Rightarrow \mathcal{J}(v) \text{ is instance of a subclass of } C & \text{(property of } \texttt{scalac}\text{)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} C & \text{(property of the JVM)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} |T| & \text{(erasure of Java class types)}
\end{array}$$

Case 2 The second case is very similar to the first, and corresponds to T being the type of a Java interface I.

$$\begin{array}{lr}
\texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} T & \\
\Rightarrow \texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} I & \text{(value of } T\text{)} \\
\Rightarrow v \text{ implements a sub-interface of } I & \text{(property of Scala)} \\
\Rightarrow \mathcal{J}(v) \text{ implements a sub-interface of } I & \text{(property of } \texttt{scalac}\text{)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} I & \text{(property of the JVM)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} |T| & \text{(erasure of Java interface types)}
\end{array}$$

Case 3 The third case is when T is the type of a Scala class $P\#C[T_1, \ldots, T_n]$.

$$\begin{array}{lr}
\texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} T & \\
\Rightarrow \texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} P\#C[T_1, \ldots, T_n] & \text{(value of } T\text{)} \\
\Rightarrow v \text{ implements a sub-interface of } P\$C & \text{(property of } \texttt{scalac}\text{)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} P\$C & \text{(property of the JVM)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} |T| & \text{(erasure of Java class types)}
\end{array}$$

Case 4 The fourth case is when T is a compound type of the following form: T_1 with $\ldots$ with T_n $\{ R \}$.

$$\begin{array}{lr}
\texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} T & \\
\Rightarrow \texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} T_1 \text{ with } \ldots \text{ with } T_n \; \{ R \} & \text{(value of } T\text{)} \\
\Rightarrow \texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} T_1 & \text{(property of } <:_{\mathrm{S}}\text{)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} |T_1| & \text{(case 3 above)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} |T| & \text{(definition of } |\cdot|\text{)}
\end{array}$$

Case 5 The fifth case is when T is a singleton type of the form x.type. Before examining this case, we recall a property of singleton types, which is that the type x.type is always a subtype of the type of x:

$$x.\text{type} <:_{\mathrm{S}} \texttt{typeof}_{\mathrm{S}}(x) \tag{4.3}$$

This property enables us to proceed as follows:

$$\begin{array}{lr}
\texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} T & \\
\Rightarrow \texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} x.\mathrm{type} & \text{(value of } T\text{)} \\
\Rightarrow \texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} \texttt{typeof}_{\mathrm{S}}(x) & \text{(property 4.3, transitivity of } <:_{\mathrm{S}}\text{)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} |\texttt{typeof}_{\mathrm{S}}(x)| & \text{(case 3 above)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} |T| & \text{(definition of } |\cdot|\text{)}
\end{array}$$

Case 6 The sixth case is when T is an array type $\mathrm{Array}[U]$.

$$\begin{array}{lr}
\texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} T & \\
\Rightarrow \texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} \mathrm{Array}[U] & \text{(value of } T\text{)} \\
\Rightarrow \texttt{typeof}_{\mathrm{S}}(v) \ = \ \mathrm{Array}[U] & \text{(array subtyping)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) \ = \ |U|[] & \text{(property of } \texttt{scalac}\text{)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} |T| & \text{(definition of } |\cdot|\text{)}
\end{array}$$

Case 7 The seventh case is when T is a type variable. Here, $\mathcal{B}(T)$ represents the concrete bound of T. That is, $\mathcal{B}(T)$ is the bound of T unless that bound is a type variable U, in which case $\mathcal{B}(T) = \mathcal{B}(U)$.

$$\begin{array}{lr}
\texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} T & \\
\Rightarrow \texttt{typeof}_{\mathrm{S}}(v) <:_{\mathrm{S}} \mathcal{B}(T) & \text{(bound of } T\text{)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} |\mathcal{B}(T)| & \text{(cases 1–6 above)} \\
\Rightarrow \texttt{typeof}_{\mathrm{J}}(\mathcal{J}(v)) <:_{\mathrm{J}} |T| & \text{(property of } |\cdot|\text{)}
\end{array}$$

4.3 Type representations

Knowing how the Scala compiler maps Scala types to JVM types, we can now focus our attention on the representation of Scala types as Java objects.

All Scala types are represented as subclasses of an abstract Java class called Type. The complete hierarchy is shown in figure 4.1 on the following page. The Type class declares the following methods:

```
abstract public boolean isInstance(Object o);
abstract public boolean isSameType(Type that);
abstract public boolean isSubType(Type that);

abstract public Object defaultValue();
abstract public Object[] newArray(int size);
```

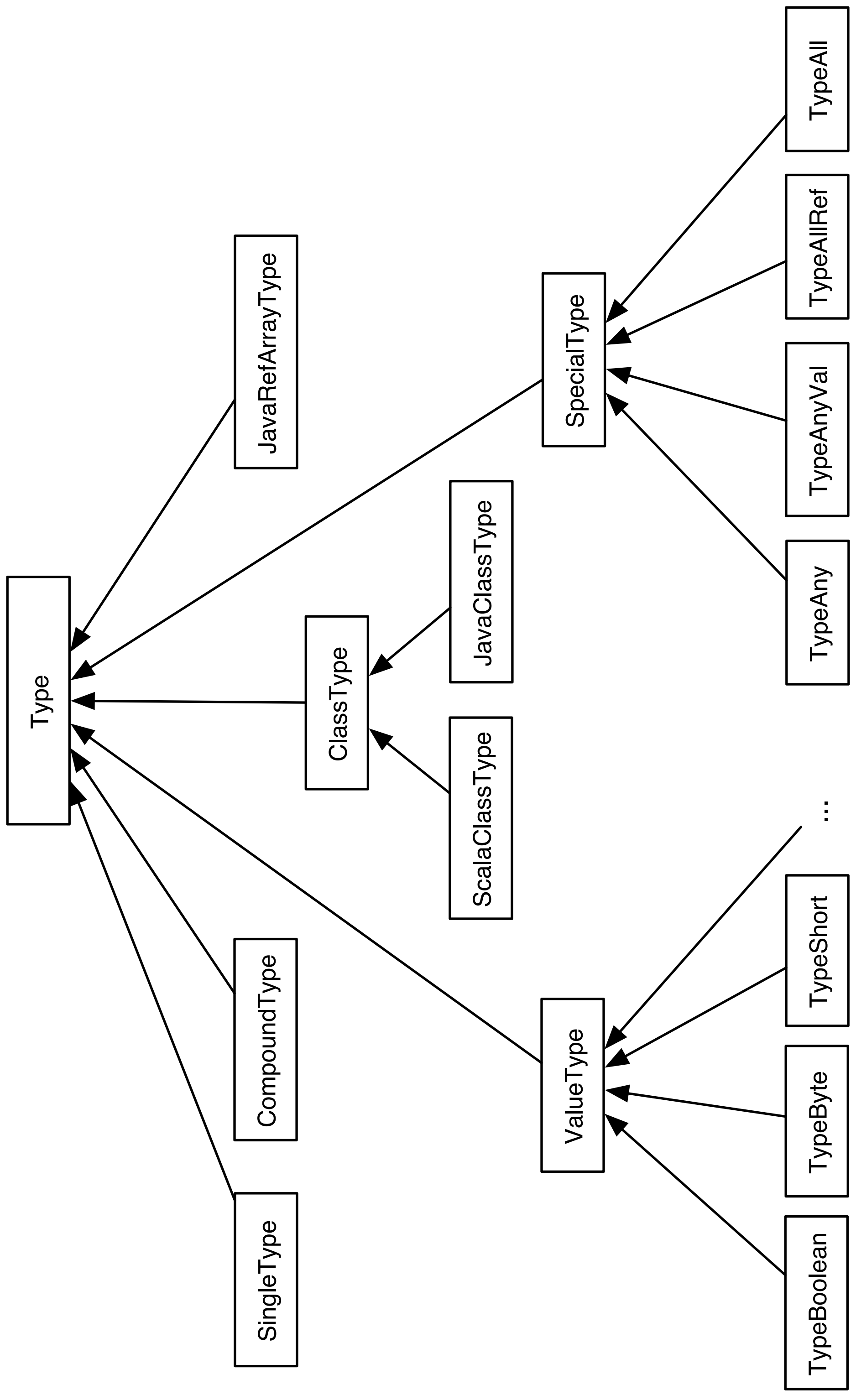


Figure 4.1: Hierarchy of Java classes representing Scala types

The `isInstance` method implements the membership test, and returns true if and only if the given object is an instance of the type to which it is applied.

Methods `isSameType` and `isSubType` implement the equivalence ($\equiv$) and subtyping ($<:_S$) relations of Scala, respectively. These relations are defined in [20, §3.5], where subtyping is called conformance.

The `defaultValue` method provides a default value appropriate for the receiver type. It is used to initialise variables, as described in section 5.9.

Finally, the `newArray` method returns a new array with the given number of elements, whose type is the receiver. It is only defined for reference types: all subclasses of `ValueType` throw an exception when this method is applied to them. This should never happen, since the compiler handles creation of arrays of value types directly as we will see in section 4.3.5.

The `scala.Type` class also provides a concrete method which implements the type cast. It is concrete because it can be expressed in terms of the `isInstance` method, as follows:

```
public Object cast(Object o) {
    if (! (o == null || isInstance(o)))
        throw new ClassCastException(this.toString());
    return o;
}
```

This code does not reveal the whole truth, however: as can be seen, if the given object passes the membership test, it is returned with type `Object`. This makes its type even more imprecise than it originally was! As we will see in section 5.8, the Scala compiler emits a CHECKCAST instruction after each call to `cast`, to make sure the code is accepted by the Java verifier.

The type cast is also used in Scala to perform conversions between numeric types, as we have seen. For that reason, the `cast` method just presented is overridden appropriately in classes representing these types.

4.3.1 A note about concurrency

An important aspect to keep in mind when designing a Java library is that Java is a concurrent language. For the library to be usable in all situations, it is therefore important to make sure it is thread-safe.

The library to manipulate run time types that we are about to describe is no exception, and care has been taken during its design to ensure thread-safety. This has been accomplished by avoiding shared state as much as possible, and using locks or atomic updates when necessary.

4.3.2 Class types

Class types represent the type of instances of a Java or Scala class. All objects existing at run time therefore have a class type attached to them. Java class types are attached to instances of Java classes, and Scala class types are attached to instances of Scala classes.

In their most general form, class types have the following form:

$$P\#C[T_1, \ldots, T_n]$$

where P is a type called the *prefix*, C is a class name, and the sequence $T_1, \ldots, T_n$ is the *instantiation*. Only the class name is always present: top-level classes have no prefix, and monomorphic classes have no instantiation. Our implementation being restricted in that non-empty prefixes have to be singleton types, we will from now on consider only class types of the form

$$p.\text{type}\#C[T_1, \ldots, T_n] \quad (\text{or, equivalently } p.C[T_1, \ldots, T_n])$$

where p is a value called the *outer instance*.

Class types which have neither a prefix nor type arguments are called *trivial (class) types*. Class types which are trivial and which have only trivial ancestors are called *strongly trivial (class) types*.

Erasure

The erasure relation transforms class types by removing their prefix and instantiation:

$$|p.C[T_1, \ldots, T_n]| = U\#C$$

where U is the erasure of the type of the prefix.

Both the prefix and the instantiation have to be preserved as data structures to provide full run time types.

Notice that trivial class types are preserved by erasure.

Subtyping

All class types, including Java class types, are subtypes of `AnyRef` and `Any`. A class type is a subtype of a compound type if it is a subtype of all the components of the compound type. Subtyping among class types is more involved, and deserves some attention. To explain it, we first define the notion of ancestors of a class type.

The *ancestors* of a class type T, written $\mathcal{A}(T)$, is the set of all the types of the base classes [20, §3.4] of T. The *non-trivial ancestors* of a class type T is the subset of the ancestors of T which are not trivial.

Example The following definitions:

```
trait Iterable[+T];
class Seq[+T] with Iterable[T];
class List[+T] extends Seq[T];
```

give rise, for example, to the following ancestors:

Type	**Ancestors**
`Seq[Object]`	`Seq[Object]`, `Iterable[Object]`, `Object`
`List[String]`	`List[String]`, `Seq[String]`, `Iterable[String]`, `Object`

In particular, notice that even though the type `List[String]` is a subtype of `Seq[Object]`, because of the covariance of the type argument, the latter is not part of the ancestors of the former.

An *instantiation* of a parametrised class type $p.C[V_1, \dots, V_n]$ is a sequence of types, written $[T_1, \dots, T_n]$, mapping variable V_i to type T_i where $i \in \{1, \dots, n\}$.

An instantiation $[T_1, \dots, T_n]$ of a parametrised class type $p.C[V_1, \dots, V_n]$ *subsumes* another instantiation $[U_1, \dots, U_n]$ if and only if the following holds:

$$\forall i \in I^0_C : T_i \equiv U_i \;\wedge\; \forall i \in I^-_C : U_i <:_S T_i \;\wedge\; \forall i \in I^+_C : T_i <:_S U_i$$

where I^0_C, I^-_C and I^+_C are the set of indexes of invariant, contravariant and covariant type parameters of C, respectively.

Subsumption is a ternary relation, as it relates two instantiations and a class name, and is written as follows:

$$[T_1, \dots, T_n] \underset{C}{\sqsubseteq} [U_1, \dots, U_n]$$

Given these notions, we can express subtyping among class types as follows:

$$\begin{gathered} S <:_S p.C[T_1, \dots, T_n] \\ \Updownarrow \\ \exists\, U_1, \dots, U_n : p.C[U_1, \dots, U_n] \in \mathcal{A}(S) \;\wedge\; [U_1, \dots, U_n] \underset{C}{\sqsubseteq} [T_1, \dots, T_n] \end{gathered}$$

Informally, a class type S is a subtype of another class type T if S has an ancestor with the same class and prefix as T, and if the instantiation of that ancestor subsumes the instantiation of T.

For a trivial class type T, which has neither a prefix nor type parameters, the subtyping test $S <:_S T$ reduces to the following check:

$$S <:_S T \Leftrightarrow T \in \mathcal{A}(S)$$

4.3.3 Java class types

A Java class type contains nothing more than the `java.lang.Class` object representing the class. It simply serves as a wrapper around Java's reflection API.

All Java class types are trivial, as they have neither a prefix, nor type parameters — at least until Java 1.4 which is the latest version currently supported by Scala.

4.3.4 Scala class types

Scala class types represent the type of all instances of Scala classes, which have the form $p.C[T_1, \ldots, T_n]$. Their representation is split in two parts:

1. the part common to all instantiations of the type constructor ($p.C$),
2. the part specific to the instantiation of the type ($[T_1, \ldots, T_n]$).

The common part is stored into a *type constructor* object, while the specific part is stored into the Scala class type itself. These two parts are described below.

Type constructor

The type constructor stores the information common to all instantiations of a polymorphic class type, which includes:

1. the prefix of the type, if any,
2. the `java.lang.Class` object representing the class of the type,
3. the variance of the type parameters,
4. the *level* of its class in the hierarchy,
5. the *ancestor code*, describing how to obtain the non-trivial ancestors of an instantiation of the type (see following section).

The level of a class is defined as the length of the shortest path from that class to class `scala.AnyRef` in the inheritance graph. Direct subclasses of `AnyRef` have level 1, their children have level 2, and so on.

Scala class types being relatively expensive to create, the type constructor also serves as a cache of all instantiations of itself. This cache is implemented as a purely functional red-black tree [24], and a reference points to its root. By updating this reference atomically, it is possible to manage this cache efficiently, without locking and in a thread-safe manner. Atomic update of the root can be easily achieved with the `AtomicReference` class of Java 5. For compatibility, the current implementation uses locks, but switching to a lock-free implementation is a matter of changing a single line in the code.

The use of a cache also ensures that there exists at most one object to represent every Scala class type. This makes it possible to efficiently compare these types for equality, by comparing references.

Actual Scala class type (instantiation)

Scala class types have a pointer to their type constructor, as well as all the information related to their instantiation, which includes:

1. the types which compose the instantiation,

2. the *ancestor cache*, containing the non-trivial ancestors of this type.

The types which compose the instantiation are reordered according to the variance of the corresponding variables: the types of the invariant variables are put first, followed by the ones of the contravariant variables, and finally the ones of the covariant ones. This means that the sets of indexes I^0, I^- and I^+ can be represented with a single integer, and thus that the test of instantiation subsumption can be performed more easily.

The ancestor cache contains all the non-trivial ancestors of the type, organised by level. Strongly trivial types, having only trivial ancestors, have an empty ancestor cache.

As an example, consider the hierarchy of figure 4.2 on the next page, taken from the Scala library. It gives rise to the ancestor cache presented in table 4.1 for object `Nil`. Notice that this cache contains neither an entry for `Object`, nor one for `Nil`, as both types are trivial.

The ancestor cache of a type is the union of the ancestor caches of its parents, to which the type itself is added unless it is trivial. The ancestor cache is computed lazily, and to speed its computation, parts of it is

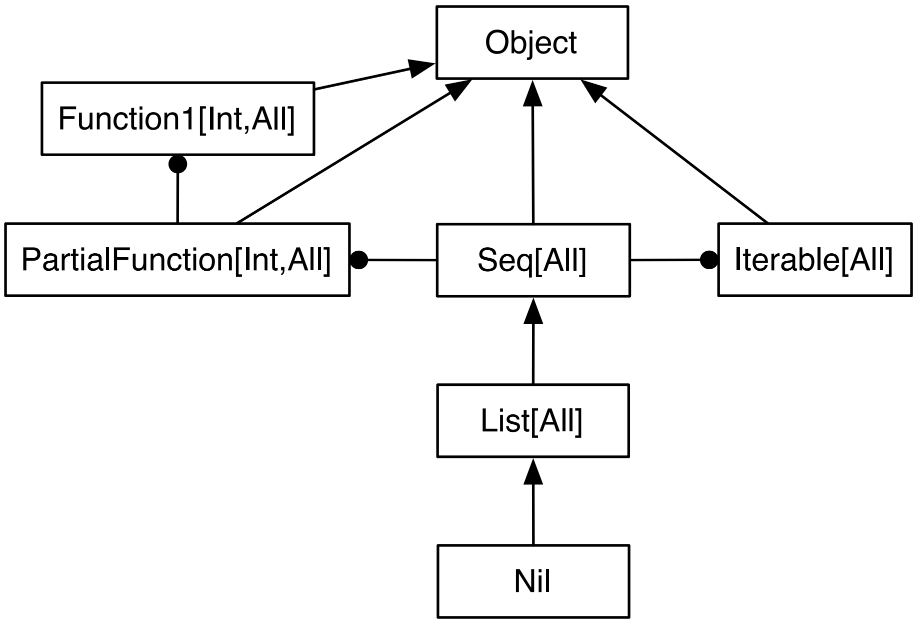


Figure 4.2: Ancestors of object `scala.Nil`

Level	**Contents**
1	`Function1[Int,All]`, `PartialFunction[Int,All]`, `Seq[All]`, `Iterable[All]`
2	`List[All]`

Table 4.1: Ancestor cache of object `scala.Nil`

performed at compile time and stored in the type constructor, as the ancestor code mentioned previously. The ancestor code describes how the ancestor cache of the first not-strongly-trivial parent of the type needs to be augmented to get the cache for the type itself.

To illustrate this, consider figure 4.3 on the facing page which shows the ancestor caches for `Seq` and its two non-trivial parents, `Iterable` and `PartialFunction`. The ancestor cache of `Seq` is obtained as follows:

1. copy the ancestor cache from parent number 0 (`PartialFunction`),
2. at level 1, add the type itself (`Seq`) as it is not trivial,
3. at level 1, add 1 entry from parent number 1 (`Iterable`), taken from its ancestor cache at offset 0, at the same level.

Steps 1 and 2 are always implicit. Step 3 is encoded as the sequence $[1, 1, 1, 0]$, which constitutes the ancestor code for `scala.Seq`.

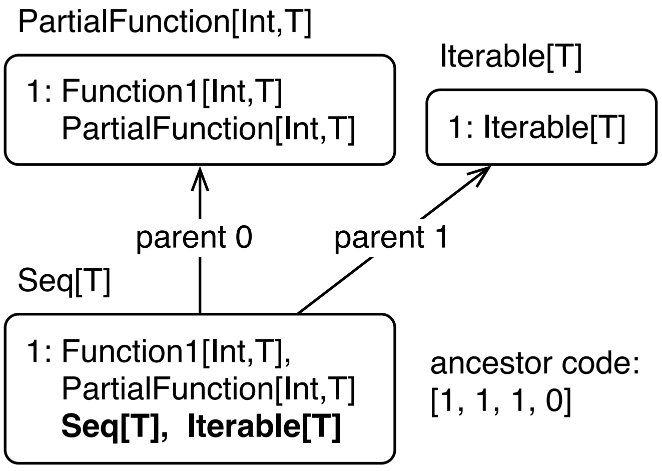


Figure 4.3: Ancestor caches for `scala.Seq` and its parents

The ancestor cache makes the implementation of the subtyping test for Scala class types relatively efficient. This implementation is given by algorithm 1 on the next page. It accesses the various components of Scala class types described above: the type constructor (constr), the ancestor cache (ancestor$[\cdot]$), the type parameters (parameter$[\cdot]$) and the indexes of the last in-, contra- and covariant type parameters (max$^{\{0,-,+\}}$).

4.3.5 Array types

Java arrays interact with run time types in non-trivial ways, mostly because they are the only parametric types recognised as such by the JVM.

In Scala, Java arrays with elements of type T are viewed as having the type `Array[`T`]`. The type constructor `Array` is restricted in the same way as in Java : it is only possible to instantiate it with a type which is either a specific value type (`Boolean`, `Int`, etc.), or a reference type. In particular, a type like `Array[`T`]` where T is a type variable bounded by `Any` is refused[2]. Arrays in Scala are invariant, unlike Java arrays which are covariant.

[2]As this limitation can be annoying for the programmer, a class `Vector` is provided in the Scala library. This class isn't limited in the types it accepts as arguments.

Algorithm 1 Subtyping test for Scala class types

```
procedure IsSubType($t_1, t_2$)                    ▷ $t_1 <:_S t_2$ for $t_1, t_2$: Scala class types
    if $\exists t'_1 : t'_1 \in t_1$.ancestor[$t_2$.level] $\wedge$ $t'_1$.constr $=$ $t_2$.constr then
        for $i \in [0, \max^0]$ do                          ▷ check invariant parameters
            if $t'_1$.parameter[$i$] $\not\equiv$ $t_2$.parameter[$i$] then
                return false
            end if
        end for
        for $i \in [\max^0 + 1, \max^-]$ do             ▷ check contravariant parameters
            if $t_2$.parameter[$i$] $\not<:_S$ $t'_1$.parameter[$i$] then
                return false
            end if
        end for
        for $i \in [\max^- + 1, \max^+]$ do                ▷ check covariant parameters
            if $t'_1$.parameter[$i$] $\not<:_S$ $t_2$.parameter[$i$] then
                return false
            end if
        end for
        return true
    else
        return false
    end if
end procedure
```

Representation of array types

An array type can be viewed as a kind of class type, which has no prefix and one type parameter. The notion of triviality can be extended to array types, by saying that the type `Array[`T`]` is trivial if and only if T is trivial. This ensures that the key property of trivial types, which is that they are preserved by erasure, is also true for trivial array types. All trivial array types are also strongly trivial, given the fact that `Array[`T`]` inherits directly from `Object`.

All trivial array types are represented as Java class types, using the technique of the Java reflection library which consists in encoding the type as the name of a class. For example, the type `Array[Array[Object]]` is represented as a Java class type with name `[[Ljava.lang.Object;`.

While appropriate for trivial array types, this technique does not work for non-trivial ones. Such types are represented by instances of a class called `JavaRefArrayType` which stores the type of the elements of the array as well as its dimensions. This class also provides the membership test appropriate for arrays, which can fail in certain cases as we will now see.

Run time type of arrays

Instances of non-trivial array types are limited in that it is not possible to extract their Scala run time type: only the JVM run time type is available for them. It is therefore not possible to distinguish the type of two arrays when their element types have the same erasure. This has two consequences: first, we prohibit the creation of arrays of singleton or compound types; second, an instance test might fail with an exception when attempted on an array type instantiated with a non-trivial class type.

Arrays of singleton or compound types are prohibited because, as soon as they are created, their run time type becomes indistinguishable from the one of other arrays. For example, an array instantiated with the compound type `C `**`with`**` D` would be represented as an array of `C` — the erasure of `C `**`with`**` D`. There is no way to distinguish such an array from a genuine array of `C`, which means that the membership test cannot provide a reliable answer. Worse, the fact that a reliable answer cannot be provided is not detectable at this point. We therefore attack the problem at its root by failing as soon as the creation of such an array is attempted.[3] Notice that this failure can happen either at compilation time, or at run time, as the type system of Scala does not enable us to express the constraint that a

[3]We do provide a way to circumvent this restriction, should this be absolutely necessary, by calling a function of the Scala run time system disabling the check.

type variable can only be instantiated with something which is neither a singleton, nor a compound type.

Arrays of non-trivial class types present a similar problem, albeit not as severe. For example, an array of a parametric type like `List[String]` is not distinguishable from an array of another instantiation of the same type constructor, say `List[Object]`. This is due to the two parametric types having the same erasure, namely `List`. This time, however, the problem can be detected when the membership test is attempted, as it is easy to find out that not enough information is available to compute the correct answer. Therefore, our implementation permits the creation of arrays of non-trivial class types, but might throw an exception when a membership test is attempted. If no exception is thrown, the correct answer is obtained, as usual.

4.3.6 Singleton types

Singleton types have the form x.type where x is a value. They are represented by the value x itself, wrapped inside an instance of the class `SingleType`.

Erasure

Singleton types are erased to the erasure of the type of their value. The identity of their value, as well as its precise type, are therefore lost. Storing only the value associated with the type is enough, as its type can be extracted using the `getScalaType` method (see §5.6).

Subtyping

Subtyping for singleton types can easily be implemented using the membership test, thanks to the following equivalence:

$$x.\mathrm{type} <:_{\mathrm{S}} T \Leftrightarrow x.\mathrm{isInstanceOf}[T]$$

4.3.7 Compound types

Compound types are composed of a set of component types $T_1, \ldots, T_n$ and a refinement R.

Erasure

Erasure of compound types loses all information about the refinement, and about all component types but the first, which is erased:

$$|T_1 \text{ with } T_2 \ \ldots \text{ with } T_n \ \{R\}| = |T_1|$$

Both the types of all components and the refinement should be represented at run time to provide full run time types.

However, for our implementation, we decided to keep only the types of the components, and to represent the refinement as a simple boolean indicating whether it is empty or not. At run time, all operations on run time types involving compound types with a non-empty refinement fail if the value of that refinement would have been needed to perform the operation.

Our decision was motivated by the cost of the implementation of refinements, which we considered to be too high compared to the benefits. Refinements are costly to implement because they represent constraints on the type of the members of a class. Supporting them fully requires the ability to represent these constraints, compare them (to implement subtyping) and test whether an object satisfies them (to implement the membership test). To illustrate this, consider the following class definitions:

```
abstract class C { type T; type U; }
abstract class D { def m: C }
```

Given these classes, a refined type like `C { type T = U }` represents instances of `C` for which type member `T` is equal to type member `U`. To perform a membership test like this one:

```
v.isInstanceOf[D { def m: C { type T = U } }];
```

the following actions must be carried out at run time:

1. first check whether value `v` is an instance of class `D`, which poses no problem,
2. then, if the first condition is satisfied, extract the (Scala) return type of method `m` from `v` — this alone requires reflection capabilities for Scala,
3. finally, check that the return type of method `m` in `v` is actually a subtype of the refined type `C { type T = U }` — this requires the comparison of constraints, for example if `v` is an instance of a class where the return type of `m` is `C { type U = T }`.

Subtyping

A compound type can be a subtype of either a class type or of an other compound type.

A compound type is a subtype of a class type if any of its components is a subtype of the class type. This is implemented simply by iterating over the components.

A compound type T_1 is a subtype of an other compound type T_2 if T_1 is a subtype of all components of T_2 and if the refinement of T_1 subsumes the one of T_2. This is implemented by first iterating over the components of T_2, and then comparing the refinements if needed.

As we have seen, refinements are represented as simple booleans indicating whether they are empty or not. It is therefore only possible to compare two refinements if at least one is empty: an empty refinement only subsumes another empty refinement, and a non-empty refinement subsumes an empty refinement. If both refinements are non-empty, the correct answer cannot be computed, and our implementation fails with an exception.

4.3.8 Value types

Value types in Scala include types corresponding to Java's primitive types (e.g. `scala.Int` corresponds to Java's **`int`**, and so on) as well as the type `scala.Unit` which is inhabited only by the unit value.

One subclass of `scala.Type` exists for each of these types, and one instance of each of these classes is stored in a static field of `scala.Type`. This makes the equivalence and subtyping tests easy to implement for them.

As explained in Section 3.2.3, the `asInstanceOf` method is also used in Scala to convert numeric values between the various numeric types. To support this behaviour, the `cast` methods of all classes representing numeric types contain code performing these coercions when appropriate.

Erasure

Value types are erased to themselves.

Subtyping

All value types are only subtypes of themselves, `AnyVal` and `Any`. The subtyping test is implemented trivially, by conjunction of these three possibilities.

4.3.9 Special types

Scala provides a few "special" types which are neither class types nor value types: `Any`, `AnyVal`, `All` and `AllRef`. They are handled in a similar fashion as value types, in that one instance of each of them is stored in a static field of `scala.Type`.

The type `AnyRef` is simply an alias for `Object`, hence it is always represented as the corresponding Java class type.

Erasure

All special types are erased to `Object`. This means that all information about them is lost in the process.

Subtyping

The subtyping test for special types simply follows the Scala rules:

$$\begin{aligned}
\texttt{Any} <:_S T &\Leftrightarrow T \equiv \texttt{Any} \\
\texttt{AnyVal} <:_S T &\Leftrightarrow T \equiv \texttt{Any} \vee T \equiv \texttt{AnyVal} \\
\texttt{AllRef} <:_S T &\Leftrightarrow T <:_S \texttt{AnyRef} \\
\texttt{All} <:_S T &
\end{aligned}$$

Chapter 5

Compiling Run Time Types

5.1 Introduction

Having examined the representation of Scala types as JVM objects at run time, we consider the compilation of expressions involving types. These expressions include of course the membership tests and the type casts, but also all invocations of polymorphic methods and class constructors.

5.2 Compiling type expressions

The TypesAsValues phase takes a Scala program where all types are still present, and transforms it into one where every type expression is matched by a corresponding value expression. The types can then be (partially) erased by the Erasure phase, without preventing the correct execution of type-dependent operations at run time.

The matching of type expressions by value expressions is performed as follows:

1. all polymorphic methods (including constructors) receive one additional value parameter for each of their type parameters,

2. a type instantiation method is created for every class of the program; this method expects type arguments and produces a type representation of the class type to which it corresponds,

3. all classes receive one accessor method for each of their type members,

4. all concrete classes get one method called `getScalaType` which returns the type of the object, as an instance of the `ClassType` class described in section 4.3.2,

5. all calls to the `isInstanceOf` and `asInstanceOf` methods are transformed into equivalent code which uses type representations.

These transformations are detailed below.

5.3 Polymorphic methods and classes

All polymorphic methods are transformed to get one additional value parameter for each of their type parameters. Calls to such methods are transformed accordingly, by passing the value representation of type arguments.

Class constructors of polymorphic classes are handled like polymorphic methods.

Example translation

To illustrate the translation of polymorphic methods, consider this simple definition:

```
def twice[T](x: T): Pair[T,T] =
  new Pair[T,T](x,x);
```

This method is translated as follows, with value parameters added to the method itself as well as to the constructor of the `Pair` class:

```
def twice[T](typ$T: scala.Type, x: T): Pair[T,T] =
  new Pair[T, T](typ$T, typ$T, x, x);
```

5.4 Instantiation methods and type constructors

As explained in section 4.3.4, a type constructor object is attached to every class which is not strongly trivial. This type constructor serves two purposes: it acts as an instantiation cache, storing the types corresponding to the various instantiations of the class; and it stores the information common to all these instantiations, like the variance of the type parameters.

The type constructor object is not manipulated directly by clients of the class that want to obtain the type corresponding to an instantiation of the class. Rather, these clients call an *instantiation method*, specific to the class, passing it the actual type parameters for which an instantiation is desired. The instantiation method checks in the cache attached to the type constructor whether the appropriate instantiation already exists, and returns it if this is the case. Otherwise, a new type is built, inserted into the cache, and returned.

In the code, both the instantiation methods and the type constructor object need to be nested at the same level as the class to which they correspond. This is easy for classes which are nested into other classes: the type constructor and instantiation method are both added as members of the enclosing class. Top-level classes do not have an enclosing class, however, and in that case the Scala compiler attaches the type constructor and the instantiation method as static members of these classes.

Before being passed to an instantiation method, type parameters are reordered according to their variance: all invariant parameters are put first, followed by all contravariant ones and finally all covariant ones. This is done in order to ease the implementation of the subtyping test, as we have seen in section 4.3.4. The reordering is stable, that is parameters with the same variance keep their relative position to one another. This ensures that there exists exactly one such reordering, which is necessary in the presence of separate compilation.

To build a class type, the types of all its parents which are not strongly trivial are needed, in order to compute the ancestor cache of the type itself. However, computing these parent types eagerly leads to an infinite loop in cases like the following:

```
class C[T];
class D[T] extends C[D[D[T]]];
```

Creating the type `D[Int]` will lead to the computation of the type of its parent, which will in turn lead to the creation of type `D[D[Int]]`, and so on.

This problem was noticed by Kennedy and Syme [12], and their solution, quite naturally, is to compute the type of the parents of a class lazily. However, introducing laziness to compute the parent types of all classes would be very expensive on the JVM, because a suspended computation has to be represented either by a class or using reflection, and both are expensive. Fortunately, in Scala it is possible to identify the cases for which laziness might be necessary, and use it only in those cases.

The key observation to make is that the set of classes involved in the computation of the parents of a class type is finite, and usually known at compilation time. For the example above, although we have seen that the set of types needed to compute the parents of `D[Int]` is infinite, only five classes are involved in this computation: `C`, `D`, `Int`, `AnyRef` and `Any`. The presence of `D` itself in this list indicates the potential for an infinite recursion.

Abstract type members complicate the problem as it is not possible to know to which concrete type they will be bound at run time. This can lead to a loop in the computation of parents, as exemplified by the following program:

```
class A[T];

abstract class C {
  type U;
  class D[V] extends A[U];
}

class E extends C {
  type U = D[E];
}
```

Computing the parents of class `D[Int]` in `E`, for example, will lead to the computation of `D[E]` which will itself compute `D[E]`, etc. We deal with such cases conservatively, by considering that abstract type members can be bound to *any* class in the program.

More formally, we associate to every class C the set $\mathcal{C}(C)$ of classes needed to compute the type of its parents. Additionally, we associate to every type T the set $\mathcal{T}(T)$ of classes needed to compute the type T itself and its parents. These sets are defined as the smallest sets satisfying the following equations, where $\mathcal{F}$ represents the set of all classes appearing in the program:

$$\mathcal{C}(C) = \bigcup_{i=1}^{n} \mathcal{T}(P_i) \quad \text{where } P_1, \ldots, P_n \text{ are the parent types of } C$$

$$\mathcal{T}(T) = \begin{cases} \mathcal{T}(P) \cup \{C\} \cup \mathcal{C}(C) \cup \bigcup_{i=1}^{n} \mathcal{T}(T_i) & \text{if } T = P\#C[T_1, \ldots, T_n] \\ \bigcup_{i=1}^{n} \mathcal{T}(T_i) & \text{if } T = T_1 \text{ with } T_2 \ldots T_n \\ \mathcal{F} & \text{if } T \text{ is a type member} \\ \emptyset & \text{otherwise} \end{cases}$$

We can now define the set $\mathcal{L}$ of classes whose parent types need to be computed lazily. These are the classes which are potentially required themselves to compute the type of their parents:

$$\mathcal{L} = \{C \mid C \in \mathcal{C}(C)\}$$

In practice, $\mathcal{L}$ tends to be small: for `scalac` and the Scala standard library, it contains only 17 classes out of more than 1000. Of these 17 classes, 11 pass their own type to one of their parents, *i.e.* they are of the form `D[T]` **extends** `C[D[T]]`; the remaining 6 pass a type member to their parents, and are conservatively included.

Example translation

We present two example translations: one which does not involve recursion in the computation of the parent types, and one which does.

Eager computation of parents

To illustrate the translation of a class whose parents can be computed eagerly, we consider a class modelling cells:

```
class Cell[T](x: T) {
  def get: T = x;
}
```

It gets translated to the following code:

```
class Cell[T](typ$T: scala.Type, x: T) {
  static val tConstructor$Cell$: TypeConstructor =
    new TypeConstructor(1,        // level
                        "Cell",   // name
                        null,     // outer instance
                        1, 0, 0,  // number of in-, co- and
                                  // contravariant variables
                        2,        // depth of ancestor cache
                        null);    // ancestor code (empty)

  static def instantiate$Cell$(types: Array[Type])
                             : ScalaClassType = {
    val inst: ScalaClassType =
      tConstructor$Cell$.getInstantiation(types);
    if (inst != null)
      inst
    else
       tConstructor$Cell$
```

```
          .instantiate(types, ScalaClassType.EMPTY_ARRAY)
    };
    def get: T = x;
  };
```

The translated version contains definitions of static members, which are not legal in Scala source, but can be produced by intermediate phases of the compiler, like here.

Lazy computation of parents

To present the translation of classes whose parents have to be computed lazily, we reuse the example presented above:

```
  class C[T];
  class D[T] extends C[D[D[T]]];
```

Class C is translated normally, but class D requires special care: since its parents cannot be computed eagerly, their computation is wrapped inside a fresh subclass of LazyParents, an abstract class belonging to the Scala run time system. This class declares only one method, called force, which computes and returns the parents.

```
  class D[T](typ$T: scala.Type) extends C[D[D[T]]](/* ... */) {
    val tConstructor$D$: TypeConstructor = /* ... as above */;
    static def instantiate$D$(types: Array[Type])
                             : ScalaClassType = {
      val inst: ScalaClassType =
        tConstructor$D$.getInstantiation(types);
      if (inst != null)
        inst
      else
         tConstructor$D$.instantiate(types,
                                     new LazyParents$D(types))
    }
  }
  class LazyParents$D(types: Array[Type]) extends LazyParents {
    def force: Array[ScalaClassType] = {
      Array(instantiate$C$(
        Array(instantiate$D$(
          Array(instantiate$D$(types)))))
    }
  }
```

The `force` method will be called the first time the parents of the class are needed.

5.5 Type accessor methods

The value representation of type members of classes have to be accessible from outside of the class. Therefore, the Scala compiler adds one accessor method for every type member of every class. With such a translation scheme, the concreteness of type members directly carries over to the one of their accessor: an abstract type member gets an abstract accessor, a concrete one gets a concrete accessor. The same is true of visibility modifiers.

Notice that, in Scala, it is possible to make an abstract type member concrete using an inner class definition. One must therefore be careful that this overriding is reflected by the accessor methods. For example, the following code is a legal Scala program:

```
abstract class C { type T; }
class D extends C { class T; }
```

The inner class definition in class `D` provides a concrete value for the type member `T`. The same must be true for the corresponding accessor methods: the accessor for inner class `T` in `D` must provide a concrete implementation for the abstract accessor for `T` declared in `C`. The easiest way to ensure this is to name type accessors like instantiation methods. In the above example, this means that both the accessor method for type `T` in `C` and instantiation method for class `T` in `D` are called `instantiate$T`.

Example translation

To illustrate the addition of type accessor methods, we will reuse the small program presented above:

```
abstract class C { type T; }
class D extends C { class T; }
```

The translated version, simplified to omit the type constructor and instantiation methods for classes `C` and `D`, looks as follows:

```
abstract class C {
  def instantiate$T(types: Array[Type]): Type;
  type T;
}
```

```
class D extends C {
  val tConstructor$T: TypeConstructor =
    new TypeConstructor(/* ... */);
  override def instantiate$T(types: Array[Type]): Type =
    /* ... see previous section */
  class T;
}
```

5.6 Type reflection method

Every Scala class receives an implementation for `getScalaType`, a method declared in the `scala.ScalaObject` interface[1]. This method is declared as returning an instance of `ClassType`, as all Scala classes have by definition a class type. The implementation of this method is straightforward, and consists in a single call to the instantiation method of the class.

The `getScalaType` method is used to implement the membership test, but it is also available to user programs, providing basic reflection capabilities for Scala. It is Scala's equivalent to Java's `getClass` method.

Example translation

As explained above, the body of the `getScalaType` method consists of a single call to the instantiation method of the class to which it belongs. We illustrate this on the following class representing pairs of values:

```
class Pair[T1,T2](f: T1, s: T2) {
  def fst: T1 = f;
  def snd: T2 = s;
}
```

This class is translated as follows:

```
class Pair[T1, T2](typ$T1: Type, typ$T2: Type, f: T1, s: T2) {
  /* ... instantiation method and type constructor omitted */

  override def getScalaType(): ClassType =
    instantiate$Pair$(Array(typ$T1, typ$T2));
```

[1]`ScalaObject` is a Java interface implemented by all Scala classes, containing some support methods as well as the `getScalaType` method described here.

```
    def fst: T1 = f;
    def snd: T2 = s;
  };
```

5.7 Membership tests

There are two methods to perform membership tests in Scala: the first one is `isInstanceOf`, and was already presented; the second one is called `isInstanceOf$erased`, and performs its test on the *erasure* of the type it is given as argument. It corresponds precisely to the INSTANCEOF instruction of the JVM. This variant of the membership test is not meant to be used by the programmer, but it can be introduced by intermediate phases of the compiler.

One of the aims of the TypesAsValues phase is to transform the program it receives into one that uses exclusively the erased variant of the membership test. It achieves this by translating the calls to `isInstanceOf` into a combination of calls to its erased variant, and tests performed on the type representations, as we will now see.

In some cases, it is sufficient to perform the test on the erased type only. This is for example true when the type on which the membership test is performed is statically known to be trivial, since trivial types are preserved by erasure.

It is also possible to use static knowledge about the value on which the membership test is performed to determine that using its erased variant is equivalent[2]. To illustrate this, we can reuse the `length` function presented in section 3.2.4. After pattern matching has been translated, this function starts as follows:

```
  def length[T](l: List[T]): Int = {
    var $result: Int = _;
    if (if (l.isInstanceOf[Cons[T]]) {
          val tl: List[T] = l.asInstanceOf[Cons[T]].tl();
    ...
  }
```

Considering the definitions of classes `List` and `Cons`, and the type of `l`, one can show that if `l` is an instance of `Cons[`α`]` for some α, then it is also an instance of `Cons[T]`. In other words, using the erased version of the membership test would here be equivalent to using the normal one.

[2]This optimisation is not yet implemented in `scalac`

This is not true of all membership tests produced by the translation of pattern matching, though. For example, if we slightly change the example above so that the type parameter of `Cons` is invariant instead of covariant, the membership test cannot be performed on the erased type alone anymore: it is then possible to have values which have type `List[T]` and which are instances of `Cons[`α`]` for some α, but which are *not* instances of `Cons[T]`. An example is the value `Cons("a", Nil)`, which has type `List[Object]` and is an instance of `Cons[String]`, but not an instance of `Cons[Object]`.

Occurrences of the membership test which cannot be translated to tests on the erased type have to be translated using the type representation. We separate them in two categories, depending on the nature of the type T on which the test is performed:

1. if T is statically known to be a class type, then the test is first performed on the erased type, and if it succeeds, is performed on the full type,

2. if T is something else, then the membership test is translated using a call to the `isInstance` method of the type representation of T.

Example translation

To illustrate the translation of the membership test, we will use the following program fragment, where `obj` is a value of type `AnyRef`:

```
obj.isInstanceOf[List[Int]]
```

We are here in the case where the type is statically known to be a class type (case 1 above), and hence we can use the technique which performs the test first on the erased type, obtaining:

```
{ val temp$ = obj;
  temp$.isInstanceOf$erased[List[Int]]
    && instantiate$List$(Array(scala.Type.Int))
         .isNonTrivialInstance(temp$) }
```

The `isNonTrivialInstance` method does the same test as `isInstance`, under the assumption that the object is already known to be an instance of the erased type. This avoids checking the same thing twice.

The temporary variable `temp$` is introduced to avoid duplicating the code — and the possible side-effects — of the expression whose type has to be tested.

5.8 Type casts

Like the membership test, the type cast comes in two variants: the method `asInstanceOf` works on full types and is meant to be used by the programmer, while method `asInstanceOf$erased` works on erased types and is meant to be used by various phases of the compiler.

One case where it is used is in the translation of pattern matching. We can illustrate this again using the `length` function of section 3.2.4:

```
def length[T](l: List[T]): Int = {
  var $result: Int = _;
  if (if (l.isInstanceOf[Cons[T]]) {
        val tl: List[T] = l.asInstanceOf[Cons[T]].tl();
  ...
}
```

Since `l` is immutable, the underlined call to `asInstanceOf` obviously cannot fail, and can be replaced by a call to its erased variant. This is always true, therefore TransMatch systematically uses `asInstanceOf$erased` for its translation.

Calls to `asInstanceOf` are compiled by calling the `cast` method on the type representation, passing it the receiver object. If this method returns, then it returns the original object, with type `java.lang.Object`. In order to satisfy the JVM byte-code verifier, this result still has to be cast to the erased type, and the compiler inserts a call to `asInstanceOf$erased` in that aim.

Like for the membership test, trivial types can be handled more efficiently: the call to the `cast` method can be omitted, leaving only the call to `asInstanceOf$erased`.

Example translation

Type casts are translated in a very similar fashion as membership tests, the only difference being the erased cast inserted after the call to `cast`. This is illustrated by the following example, similar to the one of the previous section:

```
obj.asInstanceOf[List[Int]];
```

The resulting translation is:

```
instantiate$List$(Array(scala.Type.Int))
  .cast(obj).asInstanceOf$erased[List[Int]];
```

5.9 Default values

In Scala, a variable can be assigned a default value, which is written using the underscore character '_'. The exact value represented by the underscore depends on the type of the variable: for numeric types it is 0, for the boolean type it is **false**, for unit it is (), and for other types it is **null**.

When the type of the variable is known at compilation time, the compiler can easily produce the appropriate value. When the type of the variable is not known, the compiler replaces the default value by a call to the defaultValue method of the type representation.

Example translation

A class containing a variable initialised to its default value, like the following:

```
class MutableCell[T] {
  private var contents: T = _;
  def set(x: T): Unit = (contents = x);
  def get: T = contents;
}
```

is turned into the code below:

```
class MutableCell[T](typ$T: scala.Type) {
  private var contents: T = typ$T.defaultValue();
  def set(x: T): Unit = (contents = x);
  def get: T = contents;
};
```

Notice that leaving the value undefined and letting the JVM initialise it with **null** would not work for mutable cells instantiated with value types (numerical types or unit). This would result in an error on attempting to unbox that **null** value.

5.10 Array creation

In Scala, it is possible to create an array whose element type is not known at compilation time, provided that this type is bounded by AnyRef. This creates a JVM array with elements whose type is the erasure of their Scala type.

To support this, class Type contains a method called newArray which, given a size, creates an array of that size and of proper type.

Example translation

A method to create and fill an array can be defined as follows:

```
def newFilledArray[T <: AnyRef](n: Int, x: T): Array[T] = {
  val a = new Array[T](n);
  java.util.Arrays.fill(a, x);
  a
}
```

It is transformed to the code below:

```
def newFilledArray[T <: AnyRef](typ$T: scala.Type,
                                n: Int,
                                x: T): Array[T] = {
  val a = typ$T.newArray(n);
  java.util.Arrays.fill(a, x);
  a
}
```

5.11 Possible improvements

Our implementation is limited on three points: support for array types is not complete, refinements are not fully supported and prefixes cannot refer to class types.

The limitations of array types are inherent to the JVM, and there is therefore little we can do to alleviate them.

Refinements, as we have seen, would at least require full support for reflection. With that support in place, refinements could be represented as sequences of constraints on the type of the members of an object. Providing an efficient representation for these constraints, including operations to compare them and check that they are satisfied by an object, is still an open problem.

The restriction on prefixes should be the easiest to remove, and we will briefly sketch how this could be achieved.

Currently, our implementation only allows prefixes which are empty, or which are singleton types. Scala, however, permits the use of other class types as prefixes. To illustrate this limitation, we can use the following example:

```
class O[T] { class I[U] extends Pair[T,U]; }
```

While our implementation allows the creation of type `o.I[Int]` (which is the same as `o.type#I[Int]`) for some instance `o` of `O`, it does not allow the creation of type `O[String]#I[Int]`. This is due to the placement of type constructors and instantiation methods in the code: those of class `I` appear as members of class `O`, and it is not possible to access them without an instance of that class.

To lift that restriction, instantiation methods should be made static. This change alone is not sufficient, however, as these methods sometimes need access to their environment. In the example above, the instantiation method for `I` needs to access type `T` in order to compute type `Pair[T,U]`. For that reason, instantiation methods would need an additional parameter, representing their type environment.

The caller of the instantiation method would need to create and pass this type environment. It could be obtained either by calling a method on the outer instance when one is available, or constructed explicitly in the other cases.

For the example given above, this would mean that the instantiation method for `I` would appear as a static method of class `I`. It would take two arguments: a type representation for its own type parameter `U`, and the set of types representing its environment, which in this case consists only in type `T`. To build the type representation for `O[String]#I[Int]`, that method would be called with the type representations for `Int` and `String` as arguments.

A few adaptation to our run time classes (*e.g.* `TypeConstructor`) would have to be made for this solution to work, but they should not cause any particular problem.

Chapter 6

Performance impact of run time types

Run time types do not come for free: constructing them, passing them around and comparing them consumes both time and memory. In order to evaluate the cost of our implementation, we used both a micro-benchmark and several real programs. The micro-benchmark gives a good idea of the relative cost of the different aspects of run time types, while the real programs show, in practice, the impact of run time types on performance.

All measures in this chapter were taken on an iMac equipped with a 1.8 GHz PowerPC G5, and 1 GB of RAM. This machine was running version 1.4.2 of the HotSpot JVM, in client mode.

6.1 Micro benchmark

The complete listing of our micro-benchmark is given in figure 6.1. This program times the repeated invocation of several functions, each of which uses a specific aspect of the implementation of run time types. The following aspects are measured (the numbers given here correspond to those in the code and in table 6.1):

1. the benchmark overhead, measured by invoking an empty function; this serves as a baseline to judge the other measures,

2. the cost of passing precomputed types as values,

3. the cost of creating and passing types as values,

4. the cost of performing an erased membership test,

5. the cost of performing an unsuccessful membership test on a type known at compilation time — involving only an membership test on the erased type followed by a jump,

6. the cost of performing an unsuccessful membership test on a type unknown at compilation time — involving the creation of a type representation, a membership test on the erased type through Java reflection, and a jump,

7. the cost of performing a successful membership test on a type known at compilation time — involving a membership test on the erased type, the creation of a type representation, a lookup in the ancestor cache and a comparison of instantiations,

8. the cost of performing a successful membership test on a type unknown at compilation time — involving the creation of a type representation, a membership test on the erased type through Java reflection, a lookup in the ancestor cache and a comparison of instantiations.

The results are presented in table 6.1, which reports the time per iteration in nanoseconds. They show that both the creation of type representations and the membership test are very expensive operations. The relative costs of the various forms of the membership test confirm that using the erased variant as much as possible is worthwhile. Moreover, first performing the membership test on the erased type before performing it on the type representation also pays, as can be seen by comparing results 5 and 6.

Aspect	**Description**	**Time (ns)**
1	no run time types involved	14
2	precomputed type passing	19
3	type creation and passing	231
4	membership test (erased type)	31
5	unsuccessful membership test on known type	47
6	unsuccessful membership test on unknown type	417
7	successful membership test on known type	605
8	successful membership test on unknown type	651

Table 6.1: Micro benchmark timings

```
class C[T];

object MicroBenchmark {
  def empty(x: Any): Unit = ();
  def pass[T1](x: Any): Unit = ();

  def instTestErased(x: Any): Unit =
    x.isInstanceOf$erased[C[Int]];
  def instTestKnown(x: Any): Unit =
    x.isInstanceOf[C[Int]];
  def instTest[T](x: Any): Unit =
    x.isInstanceOf[T];

  def time(name: String, bench: =>Unit) = {
    System.gc();
    val begin = System.currentTimeMillis();
    var i: Int = 0;
    while (i < 10000000) { bench; i = i + 1 }
    val end = System.currentTimeMillis();
    Console.println(name + ": " + (end - begin) + " ms")
  }

  def doIt[T](x: Any, y: Any): Unit = {
    time("1 empty            ", empty(x));
    time("2 pass             ", pass[T](x));
    time("3 create + pass    ", pass[C[Int]](x));
    time("4 inst. (erased)   ", instTestErased(x));
    time("5 inst. known (no) ", instTestKnown(y));
    time("6 inst. unkn. (no) ", instTest[C[Int]](y));
    time("7 inst. known (yes)", instTestKnown(x));
    time("8 inst. unkn. (yes)", instTest[C[Int]](x));
  }

  def main(args: Array[String]): Unit =
    doIt[C[Int]](new C[Int], "a");
}
```

Figure 6.1: Micro benchmark program

6.2 Large benchmarks

To get an idea of the influence of run time types on realistic programs, we ran several of them with run time types successively enabled and disabled. We gathered some statistics about the usage of run time types, and measured their impact on code size, memory consumption and execution time.

The results are presented below, after a short introduction explaining the programs used.

6.2.1 Benchmark programs

We used five real programs as benchmarks: three (sometimes incomplete) compilers, one interpreter, and a tool to examine compiled Scala programs. All these benchmarks were written before run time types were available, and can therefore work with or without them, which was a necessary condition for our experiment.

The size of these programs is given in table 6.2. The number of lines of code mentioned includes neither blank lines nor comments. All these benchmarks also make use, at varying degrees, of the Scala library comprising about 4,500 lines of code.

	Lines of code	
Name	**Scala**	**Java**
`scalac`	17,800	33,000
`nsc`	10,400	–
`scaletta`	3,400	1,500
`scalap`	1,800	1,000
`evaluator`	260	–

Table 6.2: Size of benchmark programs

A quick description of these programs and the input they received during our measurements follows.

Scala compiler

`Scalac` is written partly in Scala and makes therefore a very good benchmark, being currently one of the biggest Scala program in existence. The part of `Scalac` written in Scala is composed of all the front-end phases, as well as the first two phases of the back-end: UnCurry and TransMatch (see figure 1.1 on page 11).

For our measures, we asked `scalac` to compile the whole source code from the `scalap` benchmark, described below.

New Scala compiler (nsc)

The Scala compiler is currently being rewritten completely in Scala. Only the scanner, parser and type checker of the new Scala compiler were complete at the time of writing, but even at this stage `nsc` constitutes a good benchmark, being relatively big.

For our measures, we gave `nsc` 50 files from the Scala standard library to parse and type-check.

Scaletta compiler

Scaletta is a small object-oriented calculus designed to formalise the core features of Scala [3]. A prototype compiler has been written in Scala by Philippe Altherr. Currently, this prototype only performs the type checking of its input, which must be included directly in the source code as an AST, as no parser has been written yet.

To measure the execution time of this prototype compiler, we gave it a small program to type-check.

Scala object file printer

`Scalap` is a tool to display the contents of compiled Scala class files. It is written almost completely in Scala, but reuses a few Java classes from `scalac`.

In this benchmark, `scalap` is given all the classes from the Scala library, and its output is redirected to `/dev/null`.

Misc evaluator

The `evaluator` benchmark is an evaluator for Misc programs. Misc (Mini Scala) is a small language developed for a compilation course at EPFL. The evaluator is a direct translation of the small-step operational semantics of the language.

The evaluator is given the recursive (and naive) definitions of the Fibonacci and factorial functions, and asked to compute these two functions applied to 10.

6.2.2 Execution statistics

To have an idea of how heavily the various benchmarks make use of run time types, we collected a few statistics. We obtained them by augmenting the code of the run time types library with assertions which never fail, but collect data as a side effect. This enabled us to easily disable them to avoid slowing down the programs when measuring execution speed.

Type representations

The first statistic of interest is the number of times a given kind of type representation is created. Moreover, it is important to know how many of these representations are different: if only a very small percentage of them are, using memoization might be interesting.

From the results presented in table 6.3, we can conclude that, at least for our programs, it makes sense to use memoization for Scala and Java class types, as we currently do. It might also be interesting for singleton and compound types, but since so few of them actually exist we decided not to memoize them.

Notice that the table doesn't include entries for value or special types, as there is always a single instance of them, cached in a static field of `scala.Type`. Also, if a particular benchmark does not create any instance of a given type representation, no data appears in the table. This is why only one row exists for Java arrays, for example: `nsc` is the only benchmark to use them.

Instance tests and casts

To know how much run time types were actually used by the various benchmarks, we counted the number of times the instance test and type cast methods were invoked. These counts do not include tests and casts performed on erased types, as these are directly translated to the appropriate JVM instructions. The results are presented in table 6.4.

These results show that type casts are a lot less frequent than membership tests. This is to be expected as most type casts are introduced by the pattern matcher, which uses the erased variant as we have seen in section 5.7.

It is also interesting to note that the non-trivial variant of the instance test is invoked much more often than the normal one. This indicates that very often membership tests are performed on class types for which the class is statically known.

Type	Benchmark	Instances	Unique
Scala class	`scalac`	79,463	180
	`nsc`	3,754,948	27,143
	`scaletta`	668,845	285
	`scalap`	59,405	1,767
	`evaluator`	321,161	52
Java class	`scalac`	200,832	20
	`nsc`	69,534	9
	`scaletta`	20,410	17
	`scalap`	79,289	2
	`evaluator`	219,087	8
Java array	`nsc`	28,640	1
singleton	`nsc`	48	2
	`scaletta`	98	5
	`scalap`	112	1
compound	`scalac`	437	19
	`nsc`	98	2
	`scaletta`	84	3

Table 6.3: Instances of various types

Benchmark	isInstance	isNonTrivialInstance	cast
`scalac`	0	28,516	0
`nsc`	5	1,232,844	7,275
`scaletta`	35	318,437	152
`scalap`	112	20,383	0
`evaluator`	0	55,782	0

Table 6.4: Calls to instance test and type cast methods

Ancestor cache searches

As described in section 4.3.4, the ancestor cache is organised as an array of lists, each level containing the list of ancestors at that level. When searching for an ancestor, a linear search is performed on the appropriate list.

To know whether the linear search was costly, we counted the number of ancestor searched, as well as the number of iterations performed. The results are presented in table 6.5.

As these results show, in all our benchmarks the ancestors were found in exactly one iteration on average, which indicates that the linear search is appropriate. Although we could construct examples where more than one iteration per search were performed, they do not seem to show up in practice.

Benchmark	**Searches**	**Iterations/searches**
`scalac`	28,516	1.0
`nsc`	1,239,208	1.0
`scaletta`	318,589	1.0
`scalap`	20,495	1.0
`evaluator`	55,782	1.0

Table 6.5: Ancestor cache searches

6.2.3 Execution time

To measure the impact of run time types on the execution speed of our benchmark programs, we compiled them first with run time types enabled, and then disabled. We executed them three times in a row and took the average execution time. All benchmarks were run with an initial heap size of 30 MB, except `nsc` which received 200 MB.

As we have seen in the previous sections, the two main operations performed on run time types are the creation of type representations and the membership test. Most of these membership tests are due to our benchmarks making heavy use of pattern matching — all but one manipulate programs represented as trees. As explained in section 5.7, almost all of these membership tests could actually be optimised and performed only the erased types. To get an idea of the importance of this optimisation, we also measured the slowdown obtained after modifying the translation of pattern matching to perform only erased membership tests. The slowdown obtained under these conditions is reported in the rightmost column of table 6.6.

The results are presented in table 6.6. The slowdown due to run time types is important, being close to 100% for nsc. For scalac, the slowdown is more modest, but this is due to it being written only partly in Scala. These results show that optimising membership tests would be worthwhile.

	Time (s)		**Slowdown**	
Benchmark	**No RTT**	**RTT**	**Normal**	**Erased**
`scalac`	9.04	10.39	15%	(13%)
`nsc`	8.12	15.11	86%	(64%)
`scaletta`	1.72	2.16	25%	(21%)
`scalap`	1.32	1.89	44%	(29%)
`evaluator`	1.17	2.04	74%	(41%)

Table 6.6: Impact of run time types on execution speed

6.2.4 Code size

To measure the growth in code size induced by run time types, we computed the total size of the class files composing them. For scalac, we chose to exclude the class files produced from Java sources, to get a more meaningful number.

The results, presented in table 6.7, indicate that the growth is relatively modest for all benchmarks but nsc. This anomaly is mostly due to the interaction between mixins and run time types. Nsc is made up of several components which are linked together using mixin composition [23]. These individual mixins contain many nested classes, for which type constructors and instantiation methods are created. When all the components are linked together using mixin composition, these new members are imported along. The code growth due to them is therefore accounted twice: once because they appear in the mixin, and once because they appear in the final composition.

6.2.5 Memory consumption

To get an idea of how memory consumption of our benchmark programs was affected by run time types, we logged the garbage collections happening while they were running. We were then able to obtain the amount of collected memory, as well as the total time spent collecting garbage.

	Code size (bytes)		
Benchmark	**No RTT**	**RTT**	**Increase**
scalac	1,704,554	2,156,291	27%
nsc	2,055,342	3,070,189	49%
scaletta	677,154	932,679	38%
scalap	303,433	376,862	24%
evaluator	96,927	117,482	21%

Table 6.7: Code size increase due to run time types

The results are presented in table 6.8. While the amount of collected memory sometimes augments in impressive proportions, the percentage of total time spent collecting garbage stays relatively constant. This can be seen in table 6.9, where garbage collection time is presented both in milliseconds and as a percentage of the total running time of table 6.6.

	Collected memory (KB)		
Benchmark	**No RTT**	**RTT**	**Increase**
scalac	47257	63502	34%
nsc	29492	135449	360%
scaletta	18522	31829	72%
scalap	4078	11529	183%
evaluator	32092	42835	33%

Table 6.8: Memory collected during benchmark runs

	GC time (ms and % of total)		
Benchmark	**No RTT**	**RTT**	**Increase**
scalac	730 (8.1%)	778 (7.5%)	7%
nsc	147 (1.8%)	403 (2.7%)	174%
scaletta	20 (1.2%)	34 (1.6%)	66%
scalap	66 (5.0%)	83 (4.4%)	26%
evaluator	10 (0.9%)	11 (0.5%)	19%

Table 6.9: Time spent collecting garbage

Chapter 7

Related Work

7.1 Mixins

Mixins originally appeared in Flavors [17] and CLOS, as a programming idiom. They were subsequently studied by Bracha [6], Bracha and Cook [7] and others.

We have seen that Scala's notion of mixin inheritance is not exactly the same as the usual one. The main difference is that mixins in Scala — like the traits of Schärli *et al.* [29, 28] — are inherited in parallel and not in sequence. Despite this difference, it is still interesting to look at existing proposals to extend Java-like languages with mixins, as the problems encountered are similar to the ones we faced with Scala.

7.1.1 Jam

Jam is an extension of Java 1.0 with mixins proposed by Ancona, Lagorio and Zucca [4]. The mixins of Jam differ from those of Scala in several respects:

1. mixins and classes are separate concepts: it is neither possible to create an instance of a mixin, nor to use a normal class as a mixin,

2. there is no way to restrict the superclasses to which a mixin can be applied (*i.e.* no equivalent to what is the superclass of a mixin in Scala): the only way for a mixin to express its dependence on existing members in the heir class is to declare them as inherited — which has the same effect as declaring them abstract in Scala,

3. mixins cannot be composed to form bigger mixins,

4. static members are allowed in mixins, which is not possible in Scala, as it doesn't have the concept of static members.

Like `scalac`, the Jam preprocessor implements mixins using code copying. The task of the Jam preprocessor is slightly simpler, however, as its source language includes neither nested classes nor type parameters.

7.1.2 MIXEDJAVA

MIXEDJAVA is an extension of CLASSICJAVA with mixins proposed by Flatt, Krishnamurthi and Felleisen [9] — CLASSICJAVA being a small subset of Java.

MIXEDJAVA is interesting for its non-standard solutions to two recurring problems of mixins: accidental override, and diamond inheritance.

To solve the problem of accidental overriding, when a conflict occurs between a member of a mixin and one of a heir, the two versions of the member are kept. When the member is then accessed through a reference, the version of the member which gets selected depends on the current *view* of the object associated with the reference. The view can be changed through explicit cast operations, method calls or subsumption.

To solve the problem of diamond inheritance, the idea is to restrict the notion of subsumption, so that if a class inherits some mixin M several times, then its instances cannot be viewed directly as instances of M. This is best illustrated with an example. Figure 7.1 shows a class C inheriting from mixin M through two different paths. In MIXEDJAVA, an instance of C cannot be viewed directly as an instance of M, because it is not clear *which* version of M to choose: the one inherited through M_1, or the one inherited through M_2? To resolve the ambiguity, such an instance has first to be viewed as an instance of M_1 or M_2 and only then as an instance of M. In other words, when several paths lead to a given mixin in the inheritance hierarchy, the path to follow must be explicitly indicated using intermediate steps.

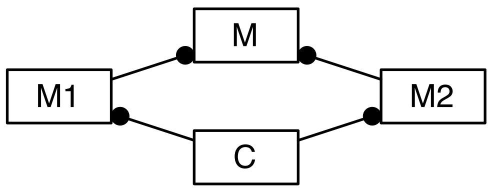


Figure 7.1: Diamond inheritance example

MIXEDJAVA was not designed as a real programming language, but rather as a calculus for a Java-like language with mixins. MIXEDJAVA itself has not been implemented, but it was inspired by its authors' work on the MzScheme system, where classes are first-class values. In such a system, mixins can be defined as normal functions operating on classes, as we have mentioned in section 2.1.3.

7.1.3 MIXGEN

MIXGEN is an extension of NEXTGEN with support for mixins proposed by Allen [2]. Mixins are supported simply by allowing a generic class to inherit from one of its type arguments. This technique is similar to the one used in C++ to implement mixins with templates [30], except that MIXGEN type-checks the mixin definitions separately, not on every application.

There is some elegance to this solution, as it is simply a generalisation (or, rather, a non-restricted form) of generic types, which does not require the existence of mixins as a separate concept. However, this technique also has several drawbacks:

1. all mixins must be generic, because they must at least have one type parameter for their superclass; this type parameter is rather artificial, and tends to pollute programs with unnecessary annotations,
2. the ability to inherit from a type parameter makes it possible to form cycles in the class hierarchy, or to make it infinite; to prevent these pathological cases, checks have to be performed during compilation,
3. mixin expansion must be performed at run time, as the set of mixin applications is potentially infinite,
4. for the same reason, accidental overriding is not detectable until run time — MIXGEN uses techniques similar to the views of MIXEDJAVA to avoid the problem altogether.

To our knowledge, MIXGEN has not yet been implemented, although implementation strategies are proposed in Allen's thesis [2, chap. 8]. The cost of the proposed techniques is hard to evaluate, but probably not negligible as a lot of work is performed at run time by the class loader.

7.1.4 Animorphic Smalltalk

The Animorphic Smalltalk system is a high performance implementation of Strongtalk, a dialect of Smalltalk which includes an optional type system and mixins as the basic unit of implementation.

The implementation of mixins in Animorphic Smalltalk is described in [5]. It uses a combination of code sharing and lazy specialisation to maximize performance: mixin methods which do not access instance variables or **super** are shared among all heirs of a mixin, while the others are specialised lazily, on their first invocation. Instance variables are thus always accessed directly, without resorting to accessor methods.

This technique combines the advantages of code copying and code sharing, since it is as fast as implementations based on copying, but not as expensive in terms of size. It is however usable only in cases where one has full control over the method table.

7.2 Run time types

Several extensions of Java with generic types and full support for run time types have been proposed recently, as well as one for .NET. The type system of all these extensions is simpler than the one of Scala, usually lacking features like type parameter variance, and singleton or compound types. Their implementation has reached a variable degree of maturity, ranging from unreleased research prototypes to working, efficient implementations.

In our description of these implementations, we adopt the nomenclature of Odersky and Wadler [22], distinguishing two kinds of translations of generics: homogeneous and heterogeneous. Homogeneous translations use the same piece of code for all instantiations of a generic class, while heterogeneous translations perform full specialisation of the code for every instantiation.

7.2.1 Heterogeneous translations

Heterogeneous translations specialise generic code for every type instantiation. Unless the source language is severely restricted, this specialisation must be performed partly at run time, because the set of all instantiations used in a program is not known at compilation time — it is potentially infinite. On the JVM this is generally implemented using a custom class loader producing specialised versions of pre-compiled class templates.

This implies that heterogeneous translations are usually not fully heterogeneous, in that polymorphic methods, unlike classes, cannot be specialised for each instantiation: the smallest unit of specialisation on the JVM is the class.

Heterogeneous translations have the potential of being slightly faster than homogeneous ones, since the code they produce is fully specialised for the type of data being manipulated. However, this positive aspect of specialisation might be cancelled by the decreased performance of the processor's instruction cache induced by the code size explosion. For that reason, specialisation is often limited to the primitive types (numbers, booleans, etc.).

Still, heterogeneous translations are interesting for run time types, as they are supported almost automatically: unlike their homogeneous counterpart, heterogeneous translations guarantee that to each different instantiation of a generic class corresponds a different translated class. Provided that the translation of types is an isomorphism, run time types for the source languages can be implemented directly using those of the JVM.

This automatic support for run time types has an additional advantage: all the complications related to arrays described in section 4.3.5 are elegantly avoided. Once again, this is due to the fact that the JVM run time type attached to the array contains enough information to recover the corresponding source type.

Agesen *et al.*

Agesen, Freund and Mitchell were the first to propose an extension of Java with parametric classes offering precise run time types [1]. Like NEXTGEN, they encode parametric types as JVM types using name mangling, without loss of information. Their extension of Java only supports parametric classes, and not parametric methods. The problem of how to translate them is thereby avoided.

Their implementation is based on a custom class loader providing specialised versions of parametric classes on demand. The type parameters of their classes can only be instantiated with reference types, and not with primitive types.

NEXTGEN

NEXTGEN, originally proposed by Cartwright and Steele [8], is based on ideas similar to the ones of Agesen *et al.* but the implementation is refined in order to reduce the cost of full specialisation. This is done by factoring out the code which is common to all instantiations of a parametric class, leaving as little code as possible to specialise.

More precisely, each parametric class is split into two classes: one contains all the code which does not depend, at run time, on the type parameters of the original class, and the other contains the remaining code in *snippet methods*. The first class is then translated homogeneously, while the second is translated heterogeneously — making NEXTGEN a kind of hybrid translator. This tremendously reduces the cost of full specialisation, as it is limited to the code which really needs run time types, typically a very small fraction of the whole.

This use of several classes to favour code sharing has one disadvantage: the subtyping relation implied by the inheritance hierarchy does not correspond to the one of the source language. To work around this problem, NEXTGEN introduces so-called *wrapper interfaces*, whose sole aim is to restore the original subtyping relation.

NEXTGEN's technique does not work for all type systems, however: as we have seen, contravariant type parameters cannot be supported since they require the extension of the subtyping relation in a way which is not compatible with the JVM.

The translation of polymorphic methods is problematic in NEXTGEN like in all heterogeneous translations. The technique described in Allen's thesis [2] is to pass *snippet environments* to polymorphic methods, which are a representation of their type arguments.

An implementation of NEXTGEN has been written by Allen, but it still has the following limitations:

1. polymorphic methods are not supported,
2. polymorphic interfaces are only partially supported,
3. separate compilation does not work.

The performance of the NEXTGEN implementation has been evaluated on relatively small benchmarks only, but the results are promising since the impact of run time types on execution time seems to be almost nonexistent.

7.2.2 Homogeneous translations

Homogeneous translations use the same piece of code for all instantiations of generic types, by erasing the types present in the original code.

As we have seen, this erasure implies that the JVM run time type attached to objects does not contain enough information to recover their source-language type. Therefore, to support full run time types, homogeneous translations have to attach additional type information to all objects. The main problems that homogeneous translations have to solve is how to represent, build, attach and compare these types. Two of these problems have received considerable attention, mostly related to their effective implementation: the building of types, and their comparison.

Several researchers have noted that instead of building type representation repeatedly, these could either be built once at the beginning of program execution, or lazily. Saha and Shao [27] implemented full lifting of type expressions to the top level, which means that all type applications are performed once and for all, when the program is started. The speed gains they report are disappointingly small, being close to 4% on average.

Full lifting of type application is only possible for languages where the set of types used by a program can be known at compilation time. While this constraint is satisfied by the SML language used by Saha and Shao — as it doesn't allow polymorphic recursion — it is violated by all extensions of Java and .NET presented here. For such languages, an approach based on lazy computation of types is appropriate, and has been used by several researchers as we will see below.

Efficient implementation of type comparisons has been studied extensively in the field of object-oriented languages, for example by Vitek *et al.* [37], and more recently by Zibin and Gil [39]. These techniques are not directly applicable to Scala, for two reasons: First, they all apply to very simple type systems, which include neither parametric types, nor singleton or compound types. Second, even though some variants of these techniques support the incremental building of the subtyping relation, they typically only permit addition of subtypes, not supertypes, to existing types. While this is sufficient for Java, it is not so for Scala which supports contravariant type parameters.

Viroli's translator

Viroli and Natali originally proposed an extension of Java with generic classes and methods quite similar to Java 5, but providing support for run time types [36]. Their implementation technique tries to lower the cost of type application by computing all the types needed by a parametric class when it is first instantiated. These types, called *friend types*, are then stored in a type descriptor object associated with the class, from which they can easily be extracted later.

In later work, Viroli also proposed an extension of his scheme to handle parametric methods [34]. Like for parametric classes, the idea is to associate with parametric methods a table of all the types they need during their execution, to avoid repeatedly building them. Dynamic dispatching makes the problem harder than for classes, though, as the caller of a method does not know statically which version of the method will be called dynamically; it cannot, therefore, know which type environment to pass. Viroli solves that problem by attaching so-called virtual parametric method tables (VPMT) to class descriptors, which associate type environments to instantiations of parametric methods.

More recently, Viroli proposed a lazy variant of his technique [33]. It differs from the previous work in that the friend types are computed lazily instead of eagerly, like in the .NET implementation presented below.

Finally, in collaboration with Cimadamore, Viroli wrote a prototype translator which uses the techniques he developed [35]. The compiler, called EGO, is an extension of Sun Microsystem's `javac` compiler, version 1.5. The programs produced by this translator create and pass type representations around, but do not use them to perform membership tests or type casts. Moreover, the issue of concurrency has been ignored.

Despite these limitations, it is interesting to compare the performance impact of the EGO compiler with ours. The benchmark used to evaluate it in [35] is Sun's `javac` compiler, and the results seem promising: the slowdown is slightly below 10%, while the code grows by about 15%. Incorporating techniques similar to the ones of EGO in `scalac` would therefore seem like a logical next step for us.

Generics for .NET

The extension of .NET with generic types proposed by Kennedy and Syme [12] differs from the ones examined until now in that, apart from the source language, it also extends the virtual machine to support generics. This gives much more freedom to the implementors, and should theoretically lead to a very efficient implementation. In fact, it is hard to imagine that one could devise a more efficient implementation based on the JVM, given the constraints it imposes.

The implementation of Kennedy and Syme is opportunistically homogeneous, in that it shares code among several instantiations only when the code happens to be the same. That is, it doesn't force all types to use a common representation in order to enable code sharing, but rather takes advantage of the sharing which happens naturally. In practice, this means that specialisation takes place for value types — integers, doubles, etc. as well as user-defined structures, unless they happen to have the same shape — while all reference types share one piece of code.

Types are passed to polymorphic classes and methods through type environments, which are filled lazily. These environments play the same role as friend types in Viroli's implementation, in that they avoid the repeated construction of type representations. They lead to an efficient implementation, since the measured slowdown due to run time types is between 10 and 20% on micro benchmarks [12].

Along with Yu, Kennedy and Syme have recently formalised the core of their implementation [38].

Chapter 8

Conclusion and Future Work

We have presented our solutions for the compilation of two important aspects of Scala: mixin inheritance and run time types. These techniques have been implemented in the Scala compiler, and evaluated on several programs of consequent size.

Mixin inheritance

We proposed two possible implementation techniques for mixin inheritance in Scala: by code copying and by delegation.

Code copying presents the advantage of incurring almost no speed penalty, as nothing distinguishes code inherited from mixins from the rest of the code. Besides, it is easy to implement even in a language which does not distinguish explicitly mixins from standard classes, like Scala. The two drawbacks of code copying are that it provides no binary compatibility unless performed at run time, and that it wastes some space.

Delegation can provide binary compatibility, but has other drawbacks: first, it requires a language in which classes and mixins are separate concepts to be realistically implementable; and second we conjecture that it would result in slower and sometimes bigger programs than code copying, because of the cost of forwarding and super-accessor methods.

For these reasons, our implementation of mixin inheritance is currently based on code copying. We evaluated the amount of code duplicated because of this in the Scala standard library, and found it to be close to 10% on average.

Run time types

We have presented our implementation of run time types for Scala, which is based on the representation of Scala types as a combination of JVM run time types and JVM objects.

While we have seen that a few other implementations of run time types exist for Java-like languages with polymorphism [12, 2, 36], they all deal with type systems that are considerably simpler than the one of Scala. None of them includes type parameter variance, virtual types, singleton types, compound types or refinements. Often, adding such features would require a major redesign of the system, or even prove impossible. For example, NEXTGEN's implementation technique is fundamentally incompatible with contravariant type parameters.

Our implementation of run time types has also reached an advanced degree of maturity, being able to successfully handle programs of several thousands of lines. Except for the limitations concerning class prefixes, refinements and arrays, the full type system of Scala is supported. Out of these three limitations, the one concerning prefixes should be easy to remove. The other two are more problematic: polymorphic arrays cannot work completely without support from the JVM, while refinements would require considerable additions to our current implementation. Further experience with refinements is needed to clearly decide whether these additions are really worth their cost.

The maturity of our implementation has enabled us to collect meaningful data about the performance of run time types. These should be useful in optimising our implementation. The main bottlenecks we have identified are the creation of type representations, and the membership test.

The first of these has already been identified by other researchers, who proposed solutions to avoid the repeated construction of type representations [27, 12, 36]. Adapting their techniques to our implementation should not be problematic, but might take some time.

The best way to address the second bottleneck is to use the membership test as little as possible. In particular, the implementation of pattern matching could be improved to use the erased version of the membership test whenever possible, and avoid it altogether when it is safe to do so, *i.e.* when matching on a sealed class.

Appendix A

Scala's type language

This appendix quickly introduces the type language of Scala using examples. Its aim is to give the reader a feeling of the different kinds of types existing in Scala, and how they can be used. More advanced examples are presented in other places [23, 19], and the Scala language specification provides all the necessary details [20].

A.1 Singleton types

Singleton types are a very basic form of dependent types [26]. The singleton type `v.type` denotes the set of values consisting of `v` and **`null`**.

As an example of the usefulness of singleton types, we can consider the problem of chaining calls to methods which have side effects. Given the following declarations:

```
class Counter {
  var value: Int = 0;
  def incr: this.type = { value = value + 1; this }
}
class DecCounter extends Counter {
  def decr: this.type = { value = value - 1; this }
}
```

it is possible to chain calls to methods `incr` and `decr`, because their return type makes it clear that they return the value **`this`** (or **`null`**). One can therefore use an instance of `DecCounter` as follows:

```
val c = new DecCounter();
Console.println(c.incr.incr.incr.decr.incr.value);
```

If the return type of `incr` was changed to `Counter`, then this would not be valid anymore, as the fact that all the invocations of `incr` above return a `DecCounter` would be lost.

A.2 Class types

Class types refer to Scala or Java classes. The type of a Java class is a simple name, but polymorphic Scala classes have type arguments, and nested ones have a prefix.

A very simple example of a polymorphic class is one representing an immutable cell containing a single object. This class is parametrized by the type of its element, as follows:

```
class Cell[T](x: T) {
  def get: T = x;
}
```

Given this definition, the type `Cell[String]` is the type of cells which contain a string.

Apart from being polymorphic, Scala classes can also be nested. For example, consider the following classes representing books and their pages:

```
class Book {
  class Page { /* ... */ };
  def addPage(p: this.Page): Unit = { /* ... */ };
}
```

The fact that class `Page` is nested inside of class `Book` means that pages from different books have a different type. For example, if we try to create two books and add a new page from the second one to the first one, we get a type error:

```
val book1 = new Book;
val book2 = new Book;
book1.addPage(new book2.Page);  // type error
```

This is due to the fact that `new book2.Page` has type `book2.type#Page` (which can also be written as `book2.Page`), while the argument passed to method `book1.addPage` must have type `book1.type#Page`. The part before the projection operator # is called the *prefix* of these types, and represents the environment in which they are nested.

A.3 Compound types

Thanks to mixin inheritance, it is possible for a class type to be simultaneously a subtype of several other class types. This fact can be expressed using a compound type of the form `C1 with C2 with ... Cn` where `C1` to `Cn` are class types.

As an example, consider a drawing program. In such a program, some objects might have a color, while others might have a position on the canvas. Each of these characteristics can be described by a trait, as follows[1]:

```
trait Colored {
  def setColor(r: Float, g: Float, b: Float): this.type;
}
trait Positioned {
  def setPosition(x: Float, y: Float): this.type;
}
```

We will suppose that the drawing program has a way to take a list of objects and spread them evenly on the canvas, while also gradually changing their color. The function implementing that feature would take as argument a list of objects which have both a color and a position. This can be expressed using the compound type `Colored with Positioned`, resulting in code similar to the following:

```
def spread(objs: List[Colored with Positioned]) = {
  for (val obj <- objs) {
    obj.setColor(...).setPosition(...);
  }
}
```

A compound type can also have a refinement, which provides more precise types for some of the members of its components. We will not explore refinements here, however.

[1]We ignore here the fact that there should also be accessors to obtain the color or position of an object.

Bibliography

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 49–65, Atlanta, GA, 1997.

[2] Eric E. Allen. *A First-Class Approach to Genericity*. PhD thesis, Rice University Computer Science Department, April 2003.

[3] Philippe Altherr and Vincent Cremet. Inner classes and virtual types. Technical report, EPFL, 2005. TR-2005013.

[4] D. Ancona, G. Lagorio, and E. Zucca. Jam – a smooth extension of Java with mixins. In *Proceedings ECOOP 2000 (European Conference on Object-Oriented Programming)*, LNCS. Springer Verlag, 2000.

[5] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, and Urs Hoelzle. Mixins in Strongtalk. In *ECOOP*, 2002.

[6] Gilad Bracha. *The programming language Jigsaw: mixins, modularity and multiple inheritance*. PhD thesis, University of Utah, Salt Lake City, UT, USA, 1992.

[7] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[8] Robert Cartwright and Guy L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), Vancouver, British Columbia*, pages 201–215. ACM, 1998.

[9] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, January 19–21, 1998.

[10] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.

[11] Bill Joy, Guy L. Steele Jr., James Gosling, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005. DRAFT.

[12] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET common language runtime. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI)*, pages 1–12, 2001.

[13] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.

[14] Xavier Leroy et al. *The Objective Caml system release 3.08*. INRIA, 2005. Available from `http://caml.inria.fr/`.

[15] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 214–223, New York, NY, 1986. ACM Press.

[16] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, April 1999.

[17] David A. Moon. Object-oriented programming with Flavors. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. ACM, 1986.

[18] Sébastien Noir. Conception et développement d'un service Web de vente aux enchères en utilisant le langage Scala, 2004.

[19] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel

Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Available from `http://scala.epfl.ch/`.

[20] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The Scala language specification. Available from `http://scala.epfl.ch/`.

[21] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, July 2003.

[22] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *24th ACM Symposium on Principles of Programming Languages*, January 1997.

[23] Martin Odersky and Matthias Zenger. Scalable component abstractions. Accepted for publication at the ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), 2005.

[24] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[25] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[26] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005.

[27] Bratin Saha and Zhong Shao. Optimal type lifting. In *Types in Compilation*, pages 156–177, 1998.

[28] Nathanael Schärli. *Traits: Composing Classes from Behavioral Building Blocks*. PhD thesis, Universität Bern, 2005.

[29] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[30] Yannis Smaragdakis and Don Batory. Mixin-based programming in C++. In *Generative and Component-Based Software Engineering Symposium (GCSE)*, pages 163–177. Springer-Verlag, LNCS 2177, 2000.

[31] Kresten Krab Thorup. Genericity in Java with virtual types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 444–471. Springer, 1997.

[32] Robert Tolksdorf. Programming languages for the Java virtual machine. `http://www.robert-tolksdorf.de/vmlanguages`.

[33] Mirko Viroli. A lazy type-passing approach for the translation of generics in Java. Unpublished draft, previously available from `http://www.ingce.unibo.it/~mviroli/LM/index.htm`.

[34] Mirko Viroli. Parametric polymorphism in Java: an efficient implementation for parametric methods. In *Selected Areas in Cryptography*, pages 610–619, 2001.

[35] Mirko Viroli. Effective and efficient compilation of run-time generics in Java. In Viviana Bono, Michele Bugliesi, and Sophia Drossopoulou, editors, *Proceedings of the 2nd Workshop on Object-Oriented Developments (WOOD 2004)*, 2004.

[36] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 146–165. ACM Press, 2000.

[37] Jan Vitek, Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, Atlanta, GA, 1997.

[38] Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of generics for the .NET Common Language Runtime. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, January 2004.

[39] Yoav Zibin and Joseph Gil. Efficient subtyping tests with PQ-encoding. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 96–107, 2001.

Curriculum Vitæ

Personal information

Name	Michel Schinz
Citizenship	Swiss (from Neuchâtel)
Date of birth	October 1st, 1973
Place of birth	Neuchâtel, Switzerland

Education

2000–2005	Ph.D., Laboratoire des Méthodes de Programmation (LAMP), EPFL, Switzerland
1991–1996	Undergraduate studies in Computer Science, EPFL
1989–1991	Gymnasium (scientific orientation C), Gymnase Cantonal de Neuchâtel

Professional experience

1998–1999	R&D engineer, Centre Suisse d'Électronique et de Microtechnique (CSEM), Neuchâtel
1997–1998	Centre Pro Natura de Champ-Pittet, Yverdon, Switzerland
1996–1997	Teaching Assistant, Laboratoire des Systèmes Répartis (LSR), EPFL