An Overview of the Scala Programming Language

Second Edition

Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos
Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud,
Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, Matthias Zenger

École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland

Technical Report LAMP-REPORT-2006-001

Abstract

Scala fuses object-oriented and functional programming in a statically typed programming language. It is aimed at the construction of components and component systems. This paper gives an overview of the Scala language for readers who are familar with programming methods and programming language design.

1 Introduction

True component systems have been an elusive goal of the software industry. Ideally, software should be assembled from libraries of pre-written components, just as hardware is assembled from pre-fabricated chips. In reality, large parts of software applications are written "from scratch", so that software production is still more a craft than an industry.

Components in this sense are simply software parts which are used in some way by larger parts or whole applications. Components can take many forms; they can be modules, classes, libraries, frameworks, processes, or web services. Their size might range from a couple of lines to hundreds of thousands of lines. They might be linked with other components by a variety of mechanisms, such as aggregation, parameterization, inheritance, remote invocation, or message passing.

We argue that, at least to some extent, the lack of progress in component software is due to shortcomings in the programming languages used to define and integrate components. Most existing languages offer only limited support for component abstraction and composition. This holds in particular for statically typed languages such as Java and C# in which much of today's component software is written.

Scala has been developed from 2001 in the programming methods laboratory at EPFL. It has been released publicly on the JVM platform in January 2004 and on the .NET platform in June 2004. A second, revised version, described in this paper was released in March 2006.

The work on Scala stems from a research effort to develop better language support for component software. There are two hypotheses that we would like to validate with the Scala experiment. First, we postulate that a programming language for component software needs to be *scalable* in the sense that the same concepts can describe small as well as large parts. Therefore, we concentrate on mechanisms for abstraction, composition, and decomposition rather than adding a large set of primitives which might be useful for components at some level of scale, but not at other levels. Second, we postulate that scalable support for components can be provided by a programming language which unifies and generalizes object-oriented and functional programming. For statically typed languages, of which Scala is an instance, these two paradigms were up to now largely separate.

To validate our hypotheses, Scala needs to be applied in the design of components and component systems. Only serious application by a user community can tell whether the concepts embodied in the language really help in the design of component software. To ease adoption by users, the new language needs to integrate well with existing platforms and components. Scala has been designed to work well with Java and C#. It adopts a large part of the syntax and type systems of these languages. At the same time, progress can sometimes only be achieved by throwing over board some existing conventions. This is why Scala is not a superset of Java. Some features are missing, others are re-interpreted to provide better uniformity of concepts.

While Scala's syntax is intentionally conventional, its type system breaks new ground in at least three areas. First, abstract type defininitions and *path-dependent types* apply the νObj calculus [36] to a concrete language design. Second, *modular mixin composition* combines the advantages of mixins and traits. Third, *views* enable component adaptation in a modular way.

The rest of this paper gives an overview of Scala. It expands on the following key aspects of the language:

- Scala programs resemble Java programs in many ways and they can seamlessly interact with code written in Java (Section 2).
- Scala has a uniform object model, in the sense that every value is an object and every operation is a method call (Section 3).

Listing 1: A simple program in Java and Scala.

```
// Java
class PrintOptions {
  public static void main(String[] args) {
    System.out.println("Options␣selected:");
    for (int i = 0; i < args.length; i++)
      if (args[i].startsWith("-"))
        System.out.println("␣"+args[i].substring(1));
  }
}

// Scala
object PrintOptions {
  def main(args: Array[String]): unit = {
    System.out.println("Options␣selected:")
    for (val arg <- args)
      if (arg.startsWith("-"))
        System.out.println("␣"+arg.substring(1))
  }
}
```

- Scala is also a functional language in the sense that functions are first-class values (Section 4).
- Scala has uniform and powerful abstraction concepts for both types and values (Section 5).
- It has flexible modular mixin-composition constructs for composing classes and traits (Section 6).
- It allows decomposition of objects by pattern matching (Section 7).
- Patterns and expressions are generalized to support the natural treatment of XML documents (Section 8).
- Scala allows external extensions of components using views (Section 9).

Section 10 discusses related work and Section 11 concludes.

This paper is intended to give a high-level overview of Scala for readers who are knowledgeable in programming languages. It is neither a precise language reference nor a tutorial. For a more precise reference, the reader is referred to the Scala Language Specification [35]. There are also several tutorials on Scala available [34, 18].

2 A Java-Like Language

Scala is designed to interact well with mainstream platforms such as Java or C#. It shares with these languages most of the basic operators, data types, and control structures.

For simplicity, we compare Scala in the following only with Java. But since Java and C# have themselves much in common, almost all similarities with Java carry over to C#. Sometimes, Scala is even closer to C# than to Java, for instance in its treatment of genericity.

Listing 1 shows a simple program in Java and Scala. The program prints out all options included in the program's command line. The example shows many similarities. Both languages use the same primitive class `String`, calling the same methods. They also use the same operators and the same conditional control construct. The example also shows some differences between the two languages. In particular:

- Scala has object definitions (starting with **`object`**) besides class definitions. An object definition defines a class with a single instance – this is sometimes called a *singleton object*. In the example above, the singleton object `PrintOptions` has `main` as a member function.
- Scala uses the *id* : *type* syntax for definitions and parameters whereas Java uses prefix types, i.e. *type id*.
- Scala's syntax is more regular than Java's in that all definitions start with a keyword. In the example above, **`def`** `main` starts a method definition.
- Scala does not require semicolons at the end of statements (they are allowed but are optional).
- Scala does not have special syntax for array types and array accesses. An array with elements of type T is written `Array[`T`]`. Here, `Array` is a standard class and $[T]$ is a type parameter. In fact, arrays in Scala inherit from functions. This is why array accesses are written like function applications $a(i)$, instead of Java's $a[i]$. Arrays are further explained in Section 4.3.
- The return type of `main` is written `unit` whereas Java uses `void`. This stems from the fact that there is no distinction in Scala between statements and expressions. Every function returns a value. If the function's right hand side is a block, the evaluation of its last expression is returned as result. The result might be the trivial value `{}` whose type is `unit`. Familar control constructs such as if-then-else are also generalized to expressions.
- Scala adopts most of Java's control structures, but it lacks Java's traditional for-statement. Instead, there are for-comprehensions which allow one to iterate directly over the elements of an array (or list, or iterator, ...) without the need for indexing. The new Java 5.0 also has a notion of "extended for-loop" which is similar to, but more restrictive than, Scala's for-comprehensions.

Even though their syntax is different, Scala programs can inter-operate without problems with Java programs. In the example above, the Scala program invokes methods `startsWith` and `substring` of `String`, which is a class defined in Java. It also accesses the static `out` field of the Java class `System`, and invokes its (overloaded) `println` method. This is possible even though Scala does not have a concept of static class members. In fact, every Java class is seen in Scala as two entities, a class containing all dynamic members and a singleton object, containing all static members. Hence, `System.out` is accessible in Scala as a member of the object `System`.

Even though it is not shown in the example, Scala classes and objects can also inherit from Java classes and implement Java interfaces. This makes it possible to use Scala code in a Java *framework*. For instance, a Scala class might be defined to implement the interface `java.util.EventListener`. Instances of that class may then be notified of events issued by Java code.

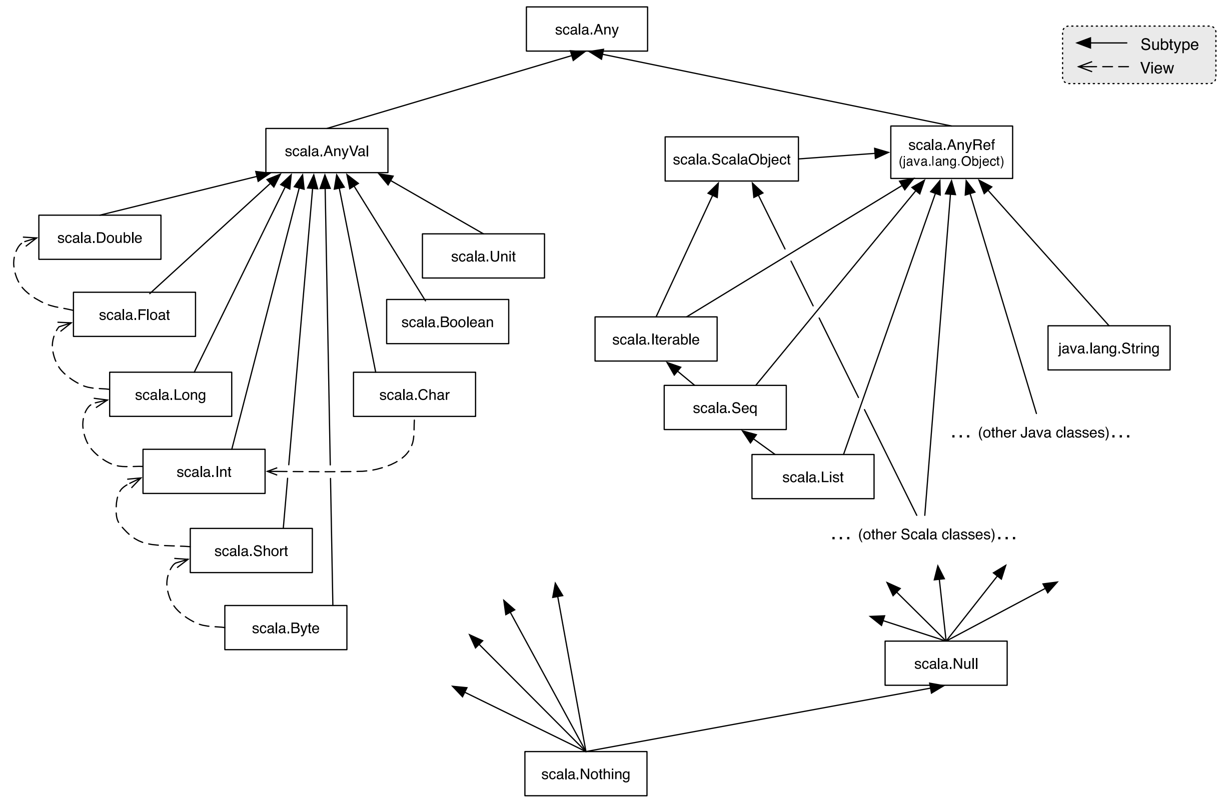

Figure 1: Class hierarchy of Scala.

3 A Unified Object Model

Scala uses a pure object-oriented model similar to Smalltalk's: Every value is an object and every operation is a message send.

3.1 Classes

Figure 2 illustrates Scala's class hierarchy. Every class in Scala inherits from class `Scala.Any`. Subclasses of `Any` fall into two categories: the *value classes* which inherit from `scala.AnyVal` and the *reference classes* which inherit from `scala.AnyRef`. Every primitive Java type name corresponds to a value class, and is mapped to it by a predefined type alias. In a Java environment, `AnyRef` is identified with the root class `java.lang.Object`. An instance of a reference class is usually implemented as a pointer to an object stored in the program heap. An instance of a value class is usually represented directly, without indirection through a pointer. Sometimes it is necessary to convert between the two representations, for example when an instance of a value class is seen as an instance of the root class `Any`. These boxing conversions and their inverses are done automatically, without explicit programmer code.

Note that the value class space is flat; all value classes are subtypes from `scala.AnyVal`, but they do not subtype each other. Instead there are views (i.e. implicit conversions, see Section 9) between elements of different value classes. We considered a design alternative with subtyping between value classes. For instance, we could have made `Int` a subtype of `Float`, instead of having an implicit conversion from `Int` to `Float`. We refrained from following this alternative, because we want to maintain the invariant that interpreting a value of a subclass as an instance of its superclass does not change the value's representation. Among other things, we want to guarantee that for each pair of types $S <: T$ and each instance x of S the following equality holds[1]:

```
x.asInstanceOf[T].asInstanceOf[S]  =  x
```

At the bottom of the type hierarchy are the two classes `scala.Null` and `scala.Nothing`. Type `Null` is a subtype of all reference types; its only instance is the **`null`** reference. Since `Null` is not a subtype of value types, **`null`** is not a member of any such type. For instance, it is not possible to assign **`null`** to a variable of type `int`.

Type `Nothing` is a subtype of every other type; there exist no instances of this type. Even though type `Nothing` is empty, it is nevertheless useful as a type parameter. For instance, the Scala library defines a value `Nil` of type `List[Nothing]`. Because lists are covariant in Scala, this makes `Nil` an instance of `List[`T`]`, for any element type T.

The equality operation == between values is designed to be transparent with respect to the type's representation. For value types, it is the natural (numeric or boolean) equality. For reference types, == is treated as an alias of the `equals` method from `java.lang.Object`. That method is originally defined as reference equality, but is meant to be overridden in subclasses to implement the natural notion of equality for these subclasses. For instance, the boxed versions of value

[1] `asInstanceOf` is Scala's standard "type cast" method defined in the root class `Any`.

types would implement an `equals` method which compares the boxed values. By contrast, in Java, == always means reference equality on reference types. While this is a bit more efficient to implement, it also introduces a serious coherence problem because boxed versions of equal values might no longer be equal with respect to ==.

Some situations require reference equality instead of user-defined equality. An example is hash-consing, where efficiency is paramount. For these cases, class `AnyRef` defines an additional `eq` method, which cannot be overridden, and is implemented as reference equality (i.e., it behaves like == in Java for reference types).

3.2 Operations

Another aspect of Scala's unified object model is that every operation is a message send, that is, the invocation of a method. For instance the addition `x + y` is interpreted as `x.+(y)`, i.e. the invocation of the method `+` with `x` as the receiver object and `y` as the method argument. This idea, which has been applied originally in Smalltalk, is adapted to the more conventional syntax of Scala as follows. First, Scala treats operator names as ordinary identifiers. More precisely, an identifier is either a sequence of letters and digits starting with a letter, or a sequence of operator characters. Hence, it is possible to define methods called `+`, `<=`, or `::`, for example. Next, Scala treats every occurrence of an identifier between two expressions as a method call. For instance, in Listing 1, one could have used the operator syntax `(arg startsWith "-")` as syntactic sugar for the more conventional syntax `(arg.startsWith("-"))`.

As an example how user-defined operators are declared and applied, consider the following implementation of a class `Nat` for natural numbers. This class (very inefficiently) represents numbers as instances of two classes `Zero` and `Succ`. The number N would hence be represented as **new** `Succ`^{N}`(Zero)`. We start the implementation with an abstract class specifying the operations supported by natural numbers. According to the definition of class `Nat`, natural numbers provide two abstract methods `isZero`, and `pred`, as well as three concrete methods `succ`, `+`, and `-`.

```
abstract class Nat {
  def isZero: boolean
  def pred: Nat
  def succ: Nat = new Succ(this)
  def + (x: Nat): Nat =
    if (x.isZero) this else succ + x.pred
  def - (x: Nat): Nat =
    if (x.isZero) this else pred - x.pred
}
```

Note that Scala allows one to define parameterless methods such as `isZero`, `pred`, and `succ` in class `Nat`. Such methods are invoked every time their name is selected; no argument list is passed. Note also that abstract class members are identified syntactically because they lack a definition; no additional **abstract** modifier is needed for them.

We now extend class `Nat` with a singleton object `Zero` and a class for representing successors, `Succ`.

```
object Zero extends Nat {
  def isZero: boolean = true
  def pred: Nat = throw new Error("Zero.pred")
  override def toString: String = "Zero"
}
```

```
class Succ(n: Nat) extends Nat {
  def isZero: boolean = false
  def pred: Nat = n
  override def toString: String = "Succ("+n+")"
}
```

The `Succ` class illustrates a difference between the class definition syntax of Scala and Java. In Scala, constructor parameters follow the class name; no separate class constructor definition within the body of `Succ` is needed. This constructor is called the *primary constructor*; the whole body of the class is executed when the primary constructor is called at the time the class is instantiated. There is syntax for *secondary constructors* in case more than one constructor is desired (see Section 5.2.1 in [35]).

The `Zero` object and the `Succ` class both implement the two abstract methods of their parent class, `Nat`. They also override the `toString` method which they inherit from class `Any`. The `override` modifier is required in Scala for methods that override a concrete method in some inherited class; it is optional for methods that implement some abstract method in their superclass. The modifier gives useful redundancy to protect against two common class of errors. One class of errors are accidental overrides, where a method in a subclass unintentionally overrides a method in a superclass. In that case the Scala compiler would complain about a missing `override` modifier. The other class of errors are broken overriding links. These arise when the parameters of a method in a superclass are changed, but one forgets to change the parameters of an overriding method in a subclass. Instead of silently converting the override to an overloading, the Scala compiler would in this case complain that the method in the subclass overrides nothing.

The ability to have user-defined infix operators raises the question about their relative precedence and associativity. One possibility would be to have "fixity"-declarations in the style of Haskell or SML, where users can declare these properties of an operator individually. However, such declarations tend to interact badly with modular programming. Scala opts for a simpler scheme with fixed precedences and associativities. The precedence of an infix operator is determined by its first character; it coincides with the operator precedence of Java for those operators that start with an operator character used in these languages. The following lists operators in increasing precedence:

(all letters)
`|`
`^`
`&`
`< >`
`= !`
`:`
`+ -`
`* / %`
(all other special characters)

Operators are usually left-associative, i.e. `x + y + z` is interpreted as `(x + y) + z`. The only exception to that rule are operators *ending* in a colon. These are treated as right-associative. An example is the list-consing operator `::`. Here, `x :: y :: zs` is interpreted as `x :: (y :: zs)`. Right-associative operators are also treated differently with respect to method lookup. Whereas normal operators take their left operand as receiver, right-associative operators take their right operand as receiver. For instance, the list consing sequence `x :: y :: zs` is treated as equivalent to `zs.::(y).::(x)`. In fact, `::` is implemented as a method in Scala's `List` class, which prefixes a given argument to the receiver list and returns the resulting list as result.

Some operators in Scala do not always evaluate their argument; examples are the standard boolean operator `&&` and `||`. Such operators can also be represented as methods because Scala allows to pass arguments by name. For instance, here is a user-defined class `Bool` that mimics the pre-defined booleans.

```
abstract class Bool {
  def && (x: => Bool): Bool
  def || (x: => Bool): Bool
}
```

In this class, the formal parameter of methods `||` and `&&` is `=> Bool`. The arrow indicates that actual arguments for these parameters are passed in unevaluated form. The arguments are evaluated every time the formal parameter name is mentioned (that is, the formal parameter behaves like a parameterless function).

Here are the two canonical instances of class `Bool`:

```
object False extends Bool {
  def && (x: => Bool): Bool = this
  def || (x: => Bool): Bool = x
}
object True extends Bool {
  def && (x: => Bool): Bool = x
  def || (x: => Bool): Bool = this
}
```

As can be seen in these implementations, the right operand of a `&&` (resp. `||`) operation is evaluated only if the left operand is the `True` (`False`) object.

As the examples in this section show, it is possible in Scala to define every operator as a method and treat every operation as an invocation of a method. In the interest of efficiency, the Scala compiler translates operations on value types directly to primitive instruction codes; this, however, is completely transparent to the programmer.

In the example above `Zero` and `Succ` both inherit from a single class. This is not the only possibility. In fact every class or object in Scala can inherit from several *traits* in addition to a normal class. A trait is an abstract class that is meant to be combined with other classes. Some traits play the role of interfaces in Java, i.e. they define a set of abstract methods which are implemented by some class. But unlike interfaces, traits in Scala can also contain method implementations or fields.

3.3 Variables and Properties

If every operation is a method invocation in Scala, what about variable dereferencing and assignment? In fact, when acting on class members these operations are also treated as method calls. For every definition of a variable **var** x: T in a class, Scala defines *setter* and *getter* methods as follows.

```
def x: T
def x_= (newval: T): unit
```

These methods reference and update a mutable memory cell, which is not accessible directly to Scala programs. Every mention of the name x in an expression is then interpreted as a call to the parameterless method x. Furthermore, every assignment x = e is interpreted as a method invocation x_=(e).

The treatment of variable accesses as method calls makes it possible to define *properties* (in the C# sense) in Scala. For instance, the following class `Celsius` defines a property `degree` which can be set only to values greater or equal than `-273`.

```
class Celsius {
  private var d: int = 0
  def degree: int = d
  def degree_=(x: int): unit = if (x >= -273) d = x
}
```

Clients can use the pair of methods defined by class `Celsius` as if it defined a variable:

```
val c = new Celsius; c.degree = c.degree - 1
```

4 Operations Are Objects

Scala is a functional language in the sense that every function is a value. It provides a lightweight syntax for the definition of anonymous and curried functions, and it also supports nested functions.

4.1 Methods are Functional Values

To illustrate the use of functions as values, consider a function `exists` that tests whether a given array has an element which satisfies a given predicate:

```
def exists[T](xs: Array[T], p: T => boolean) = {
  var i: int = 0
  while (i < xs.length && !p(xs(i))) i = i + 1
  i < xs.length
}
```

The element type of the array is arbitrary; this is expressed by the type parameter `[T]` of method `exists` (type parameters are further explained in Section 5.1). The predicate to test is also arbitrary; this is expressed by the parameter `p` of method `exists`. The type of `p` is the *function type* `T => boolean`, which has as values all functions with domain `T` and range `boolean`. Function parameters can be applied just as normal functions; an example is the application of `p` in the condition of the while-loop. Functions which take functions as arguments, or return them as results, are called *higher-order functions*.

Once we have a function `exists`, we can use it to define a function `forall` by double negation: a predicate holds for all values of an array if there does not exist an element for which the predicate does not hold. This is expressed by the following function `forall`:

```
def forall[T](xs: Array[T], p: T => boolean) = {
  def not_p(x: T) = !p(x)
  !exists(xs, not_p)
}
```

The function `forall` defines a nested function `not_p` which negates the parameter predicate `p`. Nested functions can access parameters and local variables defined in their environment; for instance `not_p` accesses `forall`'s parameter `p`.

It is also possible to define a function without giving it a name; this is used in the following shorter version of `forall`:

```
def forall[T](xs: Array[T], p: T => boolean) =
  !exists(xs, x: T => !p(x))
```

Here, `x: T => !p(x)` defines an *anonymous function* which maps its parameter `x` of type `T` to `!p(x)`.

Using `exists` and `forall`, we can define a function `hasZeroRow`, which tests whether a given two-dimensional integer matrix has a row consisting of only zeros.

```
def hasZeroRow(matrix: Array[Array[int]]) =
  exists(matrix, row: Array[int] => forall(row, 0 ==))
```

The expression `forall(row, 0 ==)` tests whether `row` consists only of zeros. Here, the == method of the number `0` is passed as argument corresponding to the predicate parameter `p`. This illustrates that methods can themselves be used as values in Scala; it is similar to the "delegates" concept in C#.

4.2 Functions are Objects

If methods are values, and values are objects, it follows that methods themselves are objects. In fact, the syntax of function types and values is just syntactic sugar for certain class types and class instances. The function type S `=>` T is equivalent to the parameterized class type `scala.Function1[`S`, `T`]`, which is defined as follows in the standard Scala library:

```
package scala
abstract class Function1[-S, +T] {
  def apply(x: S): T
}
```

Analogous conventions exist for functions with more than one argument. In general, the n-ary function type, `(`T_1`, `T_2`, ..., `T_n`) => `T is interpreted as `Function`n`[`T_1`, `T_2`, ..., `T_n`, `T`]`. Hence, functions are interpreted as objects with `apply` methods. For example, the anonymous "incrementer" function `x: int => x + 1` would be expanded to an instance of `Function1` as follows.

```
new Function1[int, int] {
  def apply(x: int): int = x + 1
}
```

Conversely, when a value of a function type is applied to some arguments, the type's `apply` method is implicitly inserted. E.g. for p of type `Function1[`S`, `T`]`, the application `p(x)` is expanded to `p.apply(x)`.

4.3 Refining Functions

Since function types are classes in Scala, they can be further refined in subclasses. An example are arrays, which are treated as special functions over the integer domain. Class `Array[T]` inherits from `Function1[int, T]`, and adds methods for array update and array length, among others:

```
package scala
class Array[T] extends Function1[int, T]
                 with Seq[T] {
  def apply(index: int): T = ...
  def update(index: int, elem: T): unit= ...
  def length: int = ...
  def exists(p: T => boolean): boolean = ...
  def forall(p: T => boolean): boolean = ...
  ...
}
```

Special syntax exists for function applications appearing on the left-hand side of an assignment; these are interpreted as applications of an `update` method. For instance, the assignment `a(i) = a(i) + 1` is interpreted as

```
a.update(i, a.apply(i) + 1) .
```

The interpretation of array accesses as function applications might seem costly. However, inlining transformations in the Scala compiler transform code such as the one above to primitive array accesses in the host system.

The above definition of the `Array` class also lists methods `exists` and `forall`. Hence, it would not have been necessary to define these operations by hand. Using the methods in class `Array`, the `hasZeroRow` function can also be written as follows.

```
def hasZeroRow(matrix: Array[Array[int]]) =
  matrix exists (row => row forall (0 ==))
```

Note the close correspondence of this code to a verbal specification of the task: "test whether in the *matrix* there *exists* a *row* such that in the *row all* elements are zeroes". Note also that we left out the type of the `row` parameter in the anonymous function. This type can be inferred by the Scala compiler from the type of `matrix.exists`.

4.4 Sequences

Higher-order methods are very common when processing sequences. Scala's library defines several different kinds of sequences including arrays, lists, streams, and iterators. All sequence types inherit from trait `scala.Seq`; and they all define a set of methods which streamlines common processing tasks. For instance, the `map` method applies a given function uniformly to all sequence elements, yielding a sequence of the function results. Another example is the `filter` method, which applies a given predicate function to all sequence elements and returns a sequence of those elements for which the predicate is true.

The application of these two functions is illustrated in the following function, `sqrts`, which takes a list `xs` of double precision numbers, and returns a list consisting of the square roots of all non-negative elements of `xs`.

```
def sqrts(xs: List[double]): List[double] =
  xs filter (0 <=) map Math.sqrt
```

Note that `Math.sqrt` comes from a Java class. Such methods can be passed to higher-order functions in the same way as methods defined in Scala.

4.5 For Comprehensions

Scala offers special syntax to express combinations of certain higher-order functions more naturally. *For comprehensions* are a generalization of list comprehensions found in languages like Haskell. With a for comprehension the `sqrts` function can be written as follows:

```
def sqrts(xs: List[double]): List[double] =
  for (val x <- xs; 0 <= x) yield Math.sqrt
```

Here, **val** x <- xs is a *generator*, which produces a sequence of values, and 0 <= x is a *filter*, which eliminates some of the produced values from consideration. The comprehension returns another sequence formed from the values produced by the **yield** part. There can be several generators and filters in a comprehension.

For comprehensions are mapped to combinations involving the higher-order methods map, flatMap, and filter. For instance, the formulation of the sqrts method above would be mapped to the previous implementation of sqrts in Section 4.4.

The power of for comprehensions comes from the fact that they are not tied to a particular data type. They can be constructed over any carrier type that defines appropriate map, flatMap, and filter methods. This includes all sequence types[2], optional values, database interfaces, as well as several other types. Scala users might apply for-comprehensions to their own types, as long as these define the required methods.

For loops are similar to comprehensions in Scala. They are mapped to combinations involving methods foreach and filter. For instance, the for loop **for** (**val** arg <- args) ... in Listing 1 is mapped to

```
args foreach (arg => ...)
```

5 Abstraction

An important issue in component systems is how to abstract from required components. There are two principal forms of abstraction in programming languages: parameterization and abstract members. The first form is typically functional whereas the second form is typically object-oriented. Traditionally, Java supported functional abstraction for values and object-oriented abstraction for operations. The new Java 5.0 with generics supports functional abstraction also for types.

Scala supports both styles of abstraction uniformly for types as well as values. Both types and values can be parameters, and both can be abstract members. The rest of this section presents both styles and reviews at the same time a large part of Scala's type system.

5.1 Functional Abstraction

The following class defines cells of values that can be read and written.

```
class GenCell[T](init: T) {
  private var value: T = init
  def get: T = value
  def set(x: T): unit = { value = x }
}
```

The class abstracts over the value type of the cell with the type parameter T. We also say, class GenCell is *generic*.

Like classes, methods can also have type parameters. The following method swaps the contents of two cells, which must both have the same value type.

```
def swap[T](x: GenCell[T], y: GenCell[T]): unit = {
  val t = x.get; x.set(y.get); y.set(t)
}
```

[2]Arrays do not yet define all of sequence methods, because some of them require run-time types, which are not yet implemented

The following program creates two cells of integers and then swaps their contents.

```
val x: GenCell[int] = new GenCell[int](1)
val y: GenCell[int] = new GenCell[int](2)
swap[int](x, y)
```

Actual type arguments are written in square brackets; they replace the formal parameters of a class constructor or method. Scala defines a sophisticated type inference system which permits to omit actual type arguments in both cases. Type arguments of a method or constructor are inferred from the expected result type and the argument types by local type inference [41, 39]. Hence, one can equivalently write the example above without any type arguments:

```
val x = new GenCell(1)
val y = new GenCell(2)
swap(x, y)
```

Parameter bounds. Consider a method updateMax which sets a cell to the maximum of the cell's current value and a given parameter value. We would like to define updateMax so that it works for all cell value types which admit a comparison function "<" defined in trait Ordered. For the moment assume this trait is defined as follows (a more refined version is in the standard Scala library).

```
trait Ordered[T] {
  def < (x: T): boolean
}
```

The updateMax method can be defined in a generic way by using bounded polymorphism:

```
def updateMax[T <: Ordered[T]](c: GenCell[T], x: T) =
  if (c.get < x) c.set(x)
```

Here, the type parameter clause [T <: Ordered[T]] introduces a bounded type parameter T. It restricts the type arguments for T to those types T that are a subtype of Ordered[T]. Therefore, the < method of class Ordered can be applied to arguments of type T. The example shows that the bounded type parameter may itself appear as part of the bound, i.e. Scala supports F-bounded polymorphism [10].

Variance. The combination of subtyping and generics in a language raises the question how they interact. If C is a type constructor and S is a subtype of T, does one also have that $C[S]$ is a subtype of $C[T]$? Type constructors with this property are called *covariant*. The type constructor GenCell should clearly not be covariant; otherwise one could construct the following program which leads to a type error at run time.

```
val x: GenCell[String] = new GenCell[String]
val y: GenCell[Any] = x;  // illegal!
y.set(1)
val z: String = y.get
```

It is the presence of a mutable variable in GenCell which makes covariance unsound. Indeed, a GenCell[String] is not a special instance of a GenCell[Any] since there are things one can do with a GenCell[Any] that one cannot do with a `GenCell[String]`; set it to an integer value, for instance.

On the other hand, for immutable data structures, covariance of constructors is sound and very natural. For instance, an immutable list of integers can be naturally seen as a special case of a list of `Any`. There are also cases where contravariance of parameters is desirable. An example are output channels `Chan[T]`, with a write operation that takes a parameter of the type parameter `T`. Here one would like to have `Chan[`S`] <: Chan[`T`]` whenever T `<:` S.

Scala allows to declare the variance of the type parameters of a class using plus or minus signs. A "+" in front of a parameter name indicates that the constructor is covariant in the parameter, a "−" indicates that it is contravariant, and a missing prefix indicates that it is non-variant.

For instance, the class `GenList` below defines a simple covariant list with methods `isEmpty`, `head`, and `tail`.

```
abstract class GenList[+T] {
  def isEmpty: boolean
  def head: T
  def tail: GenList[T]
}
```

Scala's type system ensures that variance annotations are sound by keeping track of the positions where a type parameter is used. These positions are classified as covariant for the types of immutable fields and method results, and contravariant for method argument types and upper type parameter bounds. Type arguments to a non-variant type parameter are always in non-variant position. The position flips between contra- and co-variant inside a type argument that corresponds to a contravariant parameter. The type system enforces that covariant (respectively, contravariant) type parameters are only used in covariant (contravariant) positions.

Here are two implementations of the `GenList` class:

```
object Empty extends GenList[Nothing] {
  def isEmpty: boolean = true
  def head: Nothing = throw new Error("Empty.head")
  def tail: List[Nothing] = throw new Error("Empty.tail")
}
class Cons[+T](x: T, xs: GenList[T])
        extends GenList[T] {
  def isEmpty: boolean = false
  def head: T = x
  def tail: GenList[T] = xs
}
```

Note that the `Empty` object represents the empty list for all element types. Covariance makes this possible, since `Empty`'s type, `GenList[Nothing]` is a subtype of `GenList[`T`]`, for any element type T.

Binary methods and lower bounds. So far, we have associated covariance with immutable data structures. In fact, this is not quite correct, because of *binary methods*. For instance, consider adding a `prepend` method to the `GenList` class. The most natural definition of this method takes an argument of the list element type:

```
abstract class GenList[+T] { ...
  def prepend(x: T): GenList[T] =   // illegal!
    new Cons(x, this)
}
```

However, this is not type-correct, since now the type parameter `T` appears in contravariant position inside class `GenList`. Therefore, it may not be marked as covariant. This is a pity since conceptually immutable lists should be covariant in their element type. The problem can be solved by generalizing `prepend` using a lower bound:

```
abstract class GenList[+T] { ...
  def prepend[S >: T](x: S): GenList[S] =   // OK
    new Cons(x, this)
}
```

`prepend` is now a polymorphic method which takes an argument of some supertype `S` of the list element type, `T`. It returns a list with elements of that supertype. The new method definition is legal for covariant lists since lower bounds are classified as covariant positions; hence the type parameter `T` now appears only covariantly inside class `GenList`.

It is possible to combine upper and lower bounds in the declaration of a type parameter. An example is the following method `less` of class `GenList` which compares the receiver list and the argument list.

```
abstract class GenList[+T] { ...
  def less[S >: T <: Ordered[S]](that: List[S]) =
    !that.isEmpty &&
    (this.isEmpty ||
     this.head < that.head ||
     this.head == that.head &&
       this.tail less that.tail)
}
```

The method's type parameter `S` is bounded from below by the list element type `T` and is also bounded from above by `Ordered[S]`. The lower bound is necessary to maintain covariance of `GenList`. The upper bound is needed to ensure that the list elements can be compared with the `<` operation.

Comparison with wildcards. Java 5.0 also has a way to annotate variances which is based on wildcards [45]. The scheme is essentially a syntactic variant of Igarashi and Viroli's variant parametric types [26]. Unlike in Scala, in Java 5.0 annotations apply to type expressions instead of type declarations. As an example, covariant generic lists could be expressed by writing every occurrence of the `GenList` type to match the form `GenList<? extends `T`>`. Such a type expression denotes instances of type `GenList` where the type argument is an arbitrary subtype of T.

Covariant wildcards can be used in every type expression; however, members where the type variable does not appear in covariant position are then "forgotten" in the type. This is necessary for maintaining type soundness. For instance, the type `GenCell<? extends Number>` would have just the single member `get` of type `Number`, whereas the `set` method, in which `GenCell's` type parameter occurs contravariantly, would be forgotten.

In an earlier version of Scala we also experimented with usage-site variance annotations similar to wildcards. At first-sight, this scheme is attractive because of its flexibility. A single class might have covariant as well as non-variant fragments; the user chooses between the two by placing or omitting wildcards. However, this increased flexibility comes at price, since it is now the user of a class instead of its designer who has to make sure that variance annotations are used consistently. We found that in practice it was quite difficult to achieve consistency of usage-site type annotations, so that type errors were not uncommon. By contrast, declaration-site annotations proved to be a great help in getting the design of a class right; for instance they provide excellent guidance on which methods should be generalized with lower bounds. Furthermore, Scala's mixin composition (see Section 6) makes it relatively easy to factor classes into covariant and non-variant fragments explicitly; in Java's single inheritance scheme with interfaces this would be admittedly much more cumbersome. For these reasons, later versions of Scala switched from usage-site to declaration-site variance annotations.

5.2 Abstract Members

Object-oriented abstraction can be used in Scala as an alternative to functional abstraction. For instance, here is a version of the "cell" type using object-oriented abstraction.

```
abstract class AbsCell {
  type T
  val init: T
  private var value: T = init
  def get: T = value
  def set(x: T): unit = { value = x }
}
```

The `AbsCell` class defines neither type nor value parameters. Instead it has an abstract type member `T` and an abstract value member `init`. Instances of that class can be created by implementing these abstract members with concrete definitions. For instance:

```
val cell = new AbsCell { type T = int; val init = 1 }
cell.set(cell.get * 2)
```

The type of `cell` is `AbsCell { type T = int }`. Here, the class type `AbsCell` is augmented by the *refinement* `{ type T = int }`. This makes the type alias `cell.T = int` known to code accessing the `cell` value. Therefore, type-specific operations such as the one below are legal.

```
cell.set(cell.get * 2)
```

Path-dependent types. It is also possible to access `AbsCell` without knowing the binding of its type member. For instance, the following method resets a given cell to its initial value, independently of its value type.

```
def reset(c: AbsCell): unit = c.set(c.init)
```

Why does this work? In the example above, the expression `c.init` has type `c.T`, and the method `c.set` has type `c.T => unit`. Since the formal parameter type and the argument type coincide, the method call is type-correct.

`c.T` is an instance of a *path-dependent* type. In general, such a type has the form $x_1. \ldots .x_n.t$, where $n > 0$, $x_1, \ldots, x_n$ denote immutable values and t is a type member of x_n. Path-dependent types are a novel concept of Scala; their theoretical foundation is provided by the νObj calculus [36].

Path-dependent types rely on the immutability of the prefix path. Here is an example where this immutability is violated.

```
var flip = false
def f(): AbsCell = {
  flip = !flip
  if (flip) new AbsCell { type T = int; val init = 1 }
  else new AbsCell { type T = String; val init = "" }
}
f().set(f().get)  // illegal!
```

In this example subsequent calls to `f()` return cells where the value type is alternatingly `int` and `String`. The last statement in the code above is erroneous since it tries to set an `int` cell to a `String` value. The type system does not admit this statement, because the computed type of `f().get` would be `f().T`. This type is not well-formed, since the method call `f()` is not a path.

Type selection and singleton types. In Java, where classes can also be nested, the type of a nested class is denoted by prefixing it with the name of the outer class. In Scala, this type is also expressible, in the form of `Outer # Inner`, where `Outer` is the name of the outer class in which class `Inner` is defined. The "#" operator denotes a *type selection*. Note that this is conceptually different from a path dependent type p`.Inner`, where the path p denotes a value, not a type. Consequently, the type expression `Outer #` t is not well-formed if t is an abstract type defined in `Outer`.

In fact, path dependent types in Scala can be expanded to type selections. The path dependent type $p.t$ is taken as a shorthand for p`.type #` t. Here, p`.type` is a *singleton type*, which represents just the object denoted by p. Singleton types by themselves are also useful for supporting chaining of method calls. For instance, consider a class `C` with a method `incr` which increments a protected integer field, and a subclass `D` of `C` which adds a `decr` method to decrement that field.

```
class C {
  protected var x = 0
  def incr: this.type = { x = x + 1; this }
}
class D extends C {
  def decr: this.type = { x = x - 1; this }
}
```

Then we can chain calls to the `incr` and `decr` method, as in

```
val d = new D; d.incr.decr
```

Without the singleton type **`this.type`**, this would not have been possible, since `d.incr` would be of type `C`, which does not have a `decr` member. In that sense, **`this.type`** is similar to (covariant uses of) Kim Bruce's *mytype* [9].

Family polymorphism and self types. Scala's abstract type concept is particularly well suited for modeling families of types which vary together covariantly. This concept has been called *family polymorphism*. As an example, consider the publish/subscribe design pattern. There are two classes of participants – subjects and observers. Subjects define a method `subscribe` by which observers register. They also define a `publish` method which notifies all registered observers. Notification is done by calling a method `notify` which is defined by all observers. Typically, `publish` is called when the state of a subject changes. There can be several observers associated with a subject, and an observer might observe several subjects. The `subscribe` method takes the identity of the registering observer as parameter, whereas an observer's `notify` method takes the subject that did the notification as parameter. Hence, subjects and observers refer to each other in their method signatures.

All elements of this design pattern are captured in the following system.

```
abstract class SubjectObserver {
  type S <: Subject
  type O <: Observer
  abstract class Subject requires S {
    private var observers: List[O] = List()
    def subscribe(obs: O) =
      observers = obs :: observers
    def publish =
      for (val obs <- observers) obs.notify(this)
  }
  trait Observer {
    def notify(sub: S): unit
  }
}
```

The top-level class `SubjectObserver` has two member classes: one for subjects, the other for observers. The `Subject` class defines methods `subscribe` and `publish`. It maintains a list of all registered observers in the private variable `observers`. The `Observer` trait only declares an abstract method `notify`.

Note that the `Subject` and `Observer` classes do not directly refer to each other, since such "hard" references would prevent covariant extensions of these classes in client code. Instead, `SubjectObserver` defines two abstract types `S` and `O` which are bounded by the respective class types `Subject` and `Observer`. The subject and observer classes use these abstract types to refer to each other.

Note also that class `Subject` carries an explicit `requires` annotation:

```
  abstract class Subject requires S { ...
```

The annotation expresses that `Subject` can only be instantiated as a part of a class that also conforms to `S`. Here, `S` is called a *self-type* of class `Subject`. When a self-type is given, it is taken as the type of **this** inside the class (without a self-type annotation the type of **this** is taken as usual to be the type of the class itself). In class `Subject`, the self-type is necessary to render the call `obs.notify(this)` type-correct.

Self-types can be arbitrary; they need not have a relation with the class being defined. Type soundness is still guaranteed, because of two requirements: (1) the self-type of a class must be a subtype of the self-types of all its base classes, (2) when instantiating a class in a **new** expression, it is checked that the self type of the class is a supertype of the type of the object being created.

The mechanism defined in the publish/subscribe pattern can be used by inheriting from `SubjectObserver`, defining application specific `Subject` and `Observer` classes. An example is the `SensorReader` object below that takes sensors as subjects and displays as observers.

```
object SensorReader extends SubjectObserver {
  type S = Sensor
  type O = Display
  abstract class Sensor extends Subject {
    val label: String
    var value: double = 0.0
    def changeValue(v: double) = {
      value = v
      publish
    }
  }
  abstract class Display extends Observer {
    def println(s: String) = ...
    def notify(sub: Sensor) =
      println(sub.label + "␣has␣value␣" + sub.value)
  }
}
```

In this object, type `S` is bound to `Sensor` whereas type `O` is bound to `Display`. Hence, the two formerly abstract types are now defined by overriding definitions. This "tying the knot" is always necessary when creating a concrete class instance. On the other hand, it would also have been possible to define an abstract `SensorReader` class which could be refined further by client code. In this case, the two abstract types would have been overridden again by abstract type definitions.

```
class AbsSensorReader extends SubjectObserver {
  type S <: Sensor
  type O <: Display
  ...
}
```

The following program illustrates how the `SensorReader` object is used.

```
object Test {
  import SensorReader._
  val s1 = new Sensor { val label = "sensor1" }
  val s2 = new Sensor { val label = "sensor2" }
  def main(args: Array[String]) = {
    val d1 = new Display; val d2 = new Display
    s1.subscribe(d1); s1.subscribe(d2)
    s2.subscribe(d1)
    s1.changeValue(2); s2.changeValue(3)
  }
}
```

Note the presence of an **import** clause, which makes the members of object `SensorReader` available without prefix to the code in object `Test`. Import clauses in Scala are more general than import clauses in Java. They can be used anywhere, and can import members from any object, not just from a package.

5.3 Modeling Generics with Abstract Types

The presence of two type abstraction facilities in one language raises the question of language complexity – could we have done with just one formalism? In this section we show that functional type abstraction (*aka* generics) can indeed be modeled by object-oriented type abstraction (*aka* abstract types). The idea of the encoding is as follows.

Assume you have a parameterized class C with a type parameter t (the encoding generalizes straightforwardly to multiple type parameters). The encoding has four parts, which affect the class definition itself, instance creations of the class, base class constructor calls, and type instances of the class.

1. The class definition of C is re-written as follows.

   class C {

```
    type t
    /* rest of class */
  }
```

That is, parameters of the original class are modeled as abstract members in the encoded class. If the type parameter t has lower and/or upper bounds, these carry over to the abstract type definition in the encoding. The variance of the type parameter does not carry over; variances influence instead the formation of types (see Point 4 below).

2. Every instance creation **new** $C[T]$ with type argument T is rewritten to:

   ```
   new C { type t = T }
   ```

3. If $C[T]$ appears as a superclass constructor, the inheriting class is augmented with the definition

   ```
   type t = T
   ```

4. Every type $C[T]$ is rewritten to one of the following types which each augment class C with a refinement.

`C { type t = T }`	if t is declared non-variant,
`C { type t <: T }`	if t is declared co-variant,
`C { type t >: T }`	if t is declared contra-variant.

This encoding works except for possible name-conflicts. Since the parameter name becomes a class member in the encoding, it might clash with other members, including inherited members generated from parameter names in base classes. These name conflicts can be avoided by renaming, for instance by tagging every name with a unique number.

The presence of an encoding from one style of abstraction to another is nice, since it reduces the conceptual complexity of a language. In the case of Scala, generics become simply "syntactic sugar" which can be eliminated by an encoding into abstract types. However, one could ask whether the syntactic sugar is warranted, or whether one could have done with just abstract types, arriving at a syntactically smaller language. The arguments for including generics in Scala are two-fold. First, the encoding into abstract types is not that straightforward to do by hand. Besides the loss in conciseness, there is also the problem of accidental name conflicts between abstract type names that emulate type parameters. Second, generics and abstract types usually serve distinct roles in Scala programs. Generics are typically used when one needs just type instantiation, whereas abstract types are typically used when one needs to refer to the abstract type from client code. The latter arises in particular in two situations: One might want to hide the exact definition of a type member from client code, to obtain a kind of encapsulation known from SML-style module systems. Or one might want to override the type covariantly in subclasses to obtain family polymorphism.

Could one also go the other way, encoding abstract types with generics? It turns out that this is much harder, and that it requires at least a global rewriting of the program. This was shown by studies in the domain of module systems where both kinds of abstraction are also available [27]. Furthermore in a system with bounded polymorphism, this rewriting might entail a quadratic expansion of type bounds [8]. In fact, these difficulties are not surprising if one considers the type-theoretic foundations of both systems. Generics (without F-bounds) are expressible in System $F_{<:}$ [11] whereas abstract types require systems based on dependent types. The latter are generally more expressive than the former; for instance νObj with its path-dependent types can encode $F_{<:}$.

6 Composition

After having explained Scala's constructs for type abstraction, we now focus on its constructs for class composition. Mixin class composition in Scala is a fusion of the object-oriented, linear mixin composition of Bracha [6], and the more symmetric approaches of mixin modules [14, 25] and traits [42]. To start with an example, consider the following abstraction for iterators.

```
trait AbsIterator[T] {
  def hasNext: boolean
  def next: T
}
```

Note the use of the keyword **trait** instead of **class**. A *trait* is a special form of an abstract class which does not have any value parameters for its constructor. Traits can be used in all contexts where other abstract classes appear; however only traits can be used as mixins (see below).

Next, consider a trait which extends `AbsIterator` with a method `foreach`, which applies a given function to every element returned by the iterator.

```
trait RichIterator[T] extends AbsIterator[T] {
  def foreach(f: T => unit): unit =
    while (hasNext) f(next)
}
```

Here is a concrete iterator class, which returns successive characters of a given string:

```
class StringIterator(s: String) extends AbsIterator[char] {
  private var i = 0
  def hasNext = i < s.length
  def next = { val x = s charAt i; i = i + 1; x }
}
```

Mixin-class composition We now would like to combine the functionality of `RichIterator` and `StringIterator` in a single class. With single inheritance and interfaces alone this is impossible, as both classes contain member implementations with code. Therefore, Scala provides a *mixin-class composition* mechanism which allows programmers to reuse the delta of a class definition, i.e., all new definitions that are not inherited. This mechanism makes it possible to combine `RichIterator` with `StringIterator`, as is done in the following test program. The program prints a column of all the characters of a given string.

```
object Test {
  def main(args: Array[String]): unit = {
    class Iter extends StringIterator(args(0))
               with RichIterator
    val iter = new Iter
    iter foreach System.out.println
  }
}
```

The `Iter` class in function `main` is constructed from a mixin composition of the parents `StringIterator` and `RichIterator`. The first parent is called the *superclass* of `Iter`, whereas the second parent is called a *mixin*.

Class Linearization

Mixin-class composition is a form of multiple inheritance. As such it poses several questions which do not arise for single inheritance. For instance: If several parent classes define a member with the same name, which member is inherited? To which parent member does a super-call resolve? What does it mean if a class is inherited by several different paths? In Scala, the fundamental construction for answering these questions is the class linearization.

The classes reachable through transitive closure of the direct inheritance relation from a class C are called the *base classes* of C. Because of mixins, the inheritance relationship on base classes forms in general a directed acyclic graph. The linearization $\mathcal{L}(C)$ is a total order of all base classes of C. It is computed as follows. Assume a class definition

class C **extends** B_0 **with** ... **with** B_n { ... } .

One starts with the linearization of the C's superclass B_0. This will form the last part of the linearization of C. One then prefixes to this all classes in the linearization of the first mixin class B_1, *except* those classes that are already in the constructed linearization because they were inherited from B_0. One continues like this for all other mixin classes $B_2, \ldots, B_n$, adding only those classes that are not yet in the constructed linearization. Finally, one takes C itself as first class of the linearization of C.

For instance, the linearization of class `Iter` is

```
{ Iter, RichIterator, StringIterator,
  AbsIterator, AnyRef, Any }
```

The linearization of a class refines the inheritance relation: if C is a subclass of D, then C precedes D in any linearization where both C and D occur. The linearization also satisfies the property that a linearization of a class always contains the linearization of its direct superclass as a suffix. For instance, the linearization of `StringIterator` is

```
{ StringIterator, AbsIterator, AnyRef, Any }
```

which is a suffix of the linearization of its subclass `Iter`. The same is not true for the linearization of mixin classes. It is also possible that classes of the linearization of a mixin class appear in different order in the linearization of an inheriting class, i.e. linearization in Scala is not monotonic [1].

Membership

The `Iter` class inherits members from both `StringIterator` and `RichIterator`. Generally, a class derived from a mixin composition C_n **with ... with** C_1 can define members itself and can inherit members from all parent classes. Scala adopts Java and C#'s conventions for static overloading of methods. It is thus possible that a class defines and/or inherits several methods with the same name[3]. To decide whether a defined member of a class C overrides a member of a parent class, or whether the two co-exist as overloaded variants in C, Scala uses a definition of "matching" on members, which is derived from similar concepts in Java and C#: Roughly, two members match if they have the same name, and, in case they are both methods, the same argument types.

[3]One might disagree with this design choice because of its complexity, but it is necessary to ensure interoperability, for instance when inheriting from Java's Swing libraries.

Member definitions of a class fall into two categories: concrete and abstract. There are two rules that determine the set of members of a class, one for each category:

Concrete members of a class C are all concrete definitions M in C and its base classes, except if there is already a concrete definition of a matching member in a preceding (wrt $\mathcal{L}(c)$) base class.

Abstract members of a class C are all abstract definitions M in C and its base classes, except if C has a already concrete definition of a matching member or there is already an abstract definition of a matching member in in a preceding base class.

These definitions also determines the overriding relationships between matching members of a class C and its parents. First, concrete definitions always override an abstract definitions. Second, for definitions M and M' which are both concrete or both abstract, M overrides M' if M appears in a class that precedes in the linearization of C the class in which M' is defined.

Super calls

Consider the following class of synchronized iterators, which ensures that its operations are executed in a mutually exclusive way when called concurrently from several threads.

```
abstract class SyncIterator extends AbsIterator {
  abstract override def hasNext: boolean =
    synchronized(super.hasNext)
  abstract override def next: T =
    synchronized(super.next)
}
```

To obtain rich, synchronized iterators over strings, one uses a mixin composition involving three classes:

```
StringIterator(someString) with RichIterator
                           with SyncIterator
```

This composition inherits the two members `hasNext` and `next` from the mixin class `SyncIterator`. Each method wraps a `synchronized` application around a call to the corresponding member of its superclass.

Because `RichIterator` and `StringIterator` define different sets of members, the order in which they appear in a mixin composition does not matter. In the example above, we could have equivalently written

```
StringIterator(someString) with SyncIterator
                           with RichIterator
```

There's a subtlety, however. The class accessed by the `super` calls in `SyncIterator` is not its statically declared superclass `AbsIterator`. This would not make sense, as `hasNext` and `next` are abstract in this class. Instead, *super* accesses the superclass `StringIterator` of the mixin composition in which `SyncIterator` takes part. In a sense, the superclass in a mixin composition *overrides* the statically declared superclasses of its mixins. It follows that calls to *super* cannot be statically resolved when a class is defined; their resolution has to be deferred to the point where a class is instantiated or inherited. This is made precise by the following definition.

Consider an expression **super**.M in a base class C of D. To be type correct, this expression must refer statically to some member M of a parent class of C. In the context of D, the same expression then refers to a member M' which matches M, and which appears in the first possible class that follows C in the linearization of D.

Note finally that in a language like Java or C#, the *super* calls in class SyncIterator would be illegal, precisely because they designate abstract members of the static superclass. As we have seen, Scala allows this construction, but it still has to make sure that the class is only used in a context where *super* calls access members that are concretely defined. This is enforced by the occurrence of the **abstract** and **override** modifiers in class SyncIterator. An **abstract override** modifier pair in a method definition indicates that the method's definition is not yet complete because it overrides and uses an abstract member in a superclass. A class with incomplete members must be declared abstract itself, and subclasses of it can be instantiated only once all members overridden by such incomplete members have been redefined.

Calls to *super* may be threaded so that they follow the class linearization (this is a major difference between Scala's mixin composition and multiple inheritance schemes). For example, consider another class similar to SyncIterator which prints all returned elements on standard output.

```
abstract class LoggedIterator extends AbsIterator {
  abstract override def next: T = {
    val x = super.next; System.out.println(x); x
  }
}
```

One can combine synchronized with logged iterators in a mixin composition:

```
class Iter2 extends StringIterator(someString)
               with SyncIterator with LoggedIterator;
```

The linearization of Iter2 is

```
{ Iter2, LoggedIterator, SyncIterator,
  StringIterator, AbsIterator, AnyRef, Any }
```

Therefore, class Iter2 inherits its next method from class LoggedIterator, the **super**.next call in this method refers to the next method in class SyncIterator, whose **super**.next call finally refers to the next method in class StringIterator.

If logging should be included in the synchronization, this can be achieved by reversing the order of the mixins:

```
class Iter2 extends StringIterator(someString)
               with LoggedIterator with SyncIterator;
```

In either case, calls to next follow via *super* the linearization of class Iter2.

6.1 Service-Oriented Component Model

Scala's class abstraction and composition mechanism can be seen as the basis for a *service-oriented software component model*. Software components are units of computation that *provide* a well-defined set of services. Typically, a software component is not self-contained; i.e., its service implementations rely on a set of *required services* provided by other cooperating components.

In Scala, software components correspond to classes and traits. The concrete members of a class or trait represent the provided services, deferred members can be seen as the required services. The composition of components is based on mixins, which allow programmers to create bigger components from smaller ones.

The mixin-class composition mechanism of Scala identifies services with the same name; for instance, a deferred method m can be implemented by a class C defining a concrete method m simply by mixing-in C. Thus, the component composition mechanism associates automatically required with provided services. Together with the rule that concrete class members always override deferred ones, this principle yields recursively pluggable components where component services do not have to be wired explicitly [48].

This approach simplifies the assembly of large components with many recursive dependencies. It scales well even in the presence of many required and provided services, since the association of the two is automatically inferred by the compiler. The most important advantage over traditional black-box components is that components are extensible entities: they can evolve by subclassing and overriding. They can even be used to add new services to other existing components, or to upgrade existing services of other components. Overall, these features enable a smooth incremental software evolution process.

7 Decomposition

7.1 Object-Oriented Decomposition

Often programmers have to deal with structured data. In an object-oriented language, structured data would typically be implemented by a set of classes representing the various structural constructs. For inspecting structured data, a programmer can solely rely on virtual method calls of methods provided by such classes.

Suppose we want to implement a simple evaluator for algebraic terms consisting of numbers and a binary plus operation. Using an object-oriented implementation scheme, we can decompose the evaluator according to the term structure as follows:

```
abstract class Term {
  def eval: int
}
class Num(x: int) extends Term {
  def eval: int = x
}
class Plus(left: Term, right: Term) extends Term {
  def eval: int = left.eval + right.eval
}
```

The given program models terms with the abstract class Term which defines a deferred eval method. Concrete subclasses of Term model the various term variants. Such classes have to provide concrete implementations for method eval.

Such an object-oriented decomposition scheme requires the anticipation of all operations traversing a given structure. As a consequence, even internal methods sometimes have to be exposed to some degree. Adding new methods is tedious and error-prone, because it requires all classes to be either changed or subclassed. A related problem is that implementations of operations are distributed over all participating classes making it difficult to understand and change them.

7.2 Pattern Matching Over Class Hierarchies

The program above is a good example for cases where a functional decomposition scheme is more appropriate. In a functional language, a programmer typically separates the definition of the data structure from the implementation of the operations. While data structures are usually defined using *algebraic datatypes*, operations on such datatypes are simply functions which use *pattern matching* as the basic decomposition principle. Such an approach makes it possible to implement a single `eval` function without exposing artificial auxiliary functions.

Scala provides a natural way for tackling the above programming task in a functional way by supplying the programmer with a mechanism for creating structured data representations similar to algebraic datatypes and a decomposition mechanism based on pattern matching.

Instead of adding algebraic types to the core language, Scala enhances the class abstraction mechanism to simplify the construction of structured data. Classes tagged with the **case** modifier automatically define a factory method with the same arguments as the primary constructor. Furthermore, Scala introduces pattern matching expressions in which it is possible to use such constructors of case classes as patterns. Using case classes, the algebraic term example can be implemented as follows:

```
abstract class Term
case class Num(x: int) extends Term
case class Plus(left: Term, right: Term) extends Term
```

Given these definitions, it is now possible to create the algebraic term $1+2+3$ without using the **new** primitive, simply by calling the constructors associated with case classes:

```
Plus(Plus(Num(1), Num(2)), Num(3)) .
```

Scala's pattern matching expressions provide a means of decomposition that uses these constructors as patterns. Here is the implementation of the `eval` function using pattern matching:

```
object Interpreter {
  def eval(term: Term): int = term match {
    case Num(x) => x
    case Plus(left, right) => eval(left) + eval(right)
  }
}
```

The matching expression x **match** { **case** pat_1 => e_1 **case** pat_2 => e_2 ...} matches value x against the patterns pat_1, pat_2, etc. in the given order. The program above uses patterns of the form $Constr(x_1, ..., x_n)$ where $Constr$ refers to a case class constructor and x_i denotes a variable. An object matches such a pattern if it is an instance of the corresponding case class. The matching process also instantiates the variables of the first matching pattern and executes the corresponding right-hand-side.

Such a functional decomposition scheme has the advantage that new functions can be added easily to the system. On the other hand, integrating a new case class might require changes in all pattern matching expressions. Some applications might also profit from the possibility of defining nested patterns, or patterns with guards. For instance, the pattern **case** `Plus(x, y)` **if** `x == y => ...` matches only terms of the form `t + t`. The equivalence of the two variables `x` and `y` in the previous pattern is established with the help of the guard `x == y`.

8 XML Processing

XML is a popular data format. Scala is designed to ease construction and maintenance of programs that deal with XML. It provides a data model for XML by means of traits and particular subclasses. Processing of XML can then be done by deconstructing the data using Scala's pattern matching mechanism.

8.1 Data Model

Scala's data model for XML is an immutable representation of an ordered unranked tree. In such a tree each node has a label, a sequence of children nodes, and a map from attribute keys to attribute values. This is specified in the trait `scala.xml.Node` which additionally contains equivalents of the XPath operators *child* and *descendant-or-self*, which are written \ and \\. Concrete subclasses exist for elements, text nodes, comments, processing instructions, and entity references.

XML syntax can be used directly in a Scala program, e.g., in value definitions.

```
val labPhoneBook =
  <phonebook>
  <descr>Phone numbers of<b>XML</b> hackers.</descr>
  <entry>
    <name>Burak</name>
    <phone where="work">   +41 21 693 68 67 </phone>
    <phone where="mobile"> +41 78 601 54 36 </phone>
  </entry>
  </phonebook>;
```

The value `labPhoneBook` is an XML tree; one of its nodes has the label `phone`, a child sequence consisting of a text node labeled by `+41 2..`, and a map from the attribute key `where` to the value `"work"`. Within XML syntax it is possible to escape to Scala using the brackets { and } (similar to the convention used in XQuery). For example, a `date` node with a child text node consisting of the current date can be defined by `<date>{ df.format(new java.util.Date()) }</date>`.

8.2 Schema Validation

Types of XML documents are typically specified by so called schemas. Popular schema formalisms are DTDs (Document Type Definitions) [7], XML Schema [19], and RELAX NG [33]. At this moment a simple support for DTDs is available through the `dtd2scala` tool. It converts a DTD to a set of class definitions which can only be instantiated with XML data that is valid with respect to the DTD. Existing XML documents can then be validated against the DTD by using a special load method which tries to instantiate the corresponding classes (using pattern matching). In the future, support for the richer set of types of XML Schema is planned, including static type checking through regular types.

8.3 Sequence Matching

XML nodes can be decomposed using pattern matching. Scala allows to use XML syntax here too, albeit only to match elements. The following example shows how to add an entry to a phonebook element.

```
import scala.xml.Node ;
def add(phonebook: Node, newEntry: Node): Node =
  phonebook match {
    case <phonebook>{ cs @ _* }</phonebook> =>
      <phonebook>{ cs }{ newEntry }</phonebook>
  }
val newPhoneBook =
  add(scala.xml.XML.loadFile("savedPhoneBook"),
      <entry>
        <name>Sebastian</name>
        <phone where="work">+41 21 693 68 67</phone>
      </entry>);
```

The add function performs a match on the phonebook element, binding its child sequence to the variable cs (the pattern _* matches an arbitrary sequence). Then it constructs a new phonebook element with child sequence cs followed by the node newEntry.

Sequence patterns consisting of or ending in _* extend conventional algebraic patterns discussed in Section 7. with the possibility of matching zero to arbitrary many elements of a sequence. They can be applied to any sequence, i.e. any instance of Seq[A] and to case classes that take repeated parameters. The following example illustrates their use.

```
def findRest(z: Seq[Char]): Seq[Char] = z match {
  case Seq('G', 'o', 'o', 'g', 'l', 'e',
           rest@(_*)) => rest
}
```

This pattern is used to check that a string starts with the sequence of letters "Google". If the input z matches, then the function returns what remains after the occurrence, otherwise it generates a runtime error. A previous version of Scala supported general regular expressions, but it seemed the special case described above suffices in most real life scenarios, and avoids detection and translation of top-down non-deterministic regular tree patterns, which interact badly with Scala's rich pattern language.

8.4 XML Queries through For Comprehension

A pattern match determines at most one match of a pattern. When querying XML one is often interested in locating *all* matches to a query. Scala's flexible comprehension mechanism can be used to query XML in a concise and elegant style that closely resembles XQuery. In the following example, we select all entry elements from labAddressbook and from labPhoneBook into the variables a and p, respectively. Whenever the name contents of two such entries coincide, a result element is generated which has as children the address and phone number, taken from the appropriate entry.

```
for (val a <- labAddressBook \\ "entry";
     val p <- labPhoneBook \\ "entry";
     a \ "name" == p \ "name") yield
  <result>{ a.child }{ p \ "phone" }</result>
```

9 Component Adaptation

Even component systems with powerful constructs for abstraction and composition face a problem when it comes to integrating sub-systems developed by different groups at different times. The problem is that the interface of a component developed by one group is often not quite right for clients who wish to use that component. For instance, consider a library with a class like GenList from Section 5. A client of this library might wish to treat such lists as sets, supporting operations such as member inclusion or containment tests. However, the provider of the class might not have thought of this usage scenario, and consequently might have left out these methods from the interface of GenList.

One might argue that inheritance can allow clients to tailor the supported methods of a class to their requirements; however this is only true if a client has control over all creation sites of the class. If the library also returns an operation such as

```
def fromArray(xs: Array[T]): GenList[T]
```

then inheritance cannot be used to turn a GenList into a SetList after it has been returned from method fromArray. One can circumvent this restriction to some degree by including factory methods [21] in libraries. However, this involves fairly complicated frameworks which are difficult to learn and instantiate, and it fails for library components that inherit from classes that need to be extended by clients.

This unsatisfactory situation is commonly called the *external extensibility problem*. It has been argued that this problem holds back the development of software components to a mature industry where components are independently manufactured and deployed [28].

Scala introduces a new concept to solve the external extensibility problem: *views* allow one to augment a class with new members and supported traits.

Views are a special case of implicit parameters which are themselves a useful tool for organizing rich functionality in systems. Implicit parameters let one write generic code analogous to Haskell's type classes [12], or, in a C++ context, to Siek and Lumsdaine's "concepts" [43]. Unlike with type classes, the scope of an implicit parameter can be controlled, and competing implicit parameters can coexist in different parts of one program.

Motivation As an example, let's start with an abstract class of semi-groups that support an unspecified add operation.

```
abstract class SemiGroup[a] {
  def add(x: a, y: a): a
}
```

Here's a subclass Monoid of SemiGroup which adds a unit element.

```
abstract class Monoid[a] extends SemiGroup[a] {
  def unit: a
}
```

Here are two implementations of monoids:

```
object Monoids {
  object stringMonoid extends Monoid[String] {
    def add(x: String, y: String): String = x.concat(y)
    def unit: String = ""
  }
  object intMonoid extends Monoid[int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
}
```

A sum method, which works over arbitrary monoids, can be written in plain Scala as follows.

```
def sum[a](xs: List[a])(m: Monoid[a]): a =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(m)(xs.tail)
```

One invokes this sum method by code such as:

```
sum(List("a", "bc", "def"))(Monoids.stringMonoid)
sum(List(1, 2, 3))(Monoids.intMonoid)
```

All this works, but it is not very nice. The problem is that the monoid implementations have to be passed into all code that uses them. We would sometimes wish that the system could figure out the correct arguments automatically, similar to what is done when type arguments are inferred. This is what implicit parameters provide.

Implicit Parameters: The Basics

The following slight rewrite of sum introduces m as an implicit parameter.

```
def sum[a](xs: List[a])(implicit m: Monoid[a]): a =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail))
```

As can be seen from the example, it is possible to combine normal and implicit parameters. However, there may only be one implicit parameter list for a method or constructor, and it must come last.

implicit can also be used as a modifier for definitions and declarations. Examples:

```
implicit object stringMonoid extends Monoid[String] {
  def add(x: String, y: String): String = x.concat(y)
  def unit: String = ""
}
implicit object intMonoid extends Monoid[int] {
  def add(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}
```

The principal idea behind implicit parameters is that arguments for them can be left out from a method call. If the arguments corresponding to an implicit parameter section are missing, they are inferred by the Scala compiler.

The actual arguments that are eligible to be passed to an implicit parameter of type T are all identifiers that denote an implicit definition and which satisfy either one of the following two criteria:

1. The identifier can be accessed at the point of the method call without a prefix. This includes identifiers defined locally or in some enclosing scope, as well as identifiers inherited from base classes or imported from other objects by an **import** clause.

2. Or the identifier is defined in an object C which comes with a class with the same name which is a baseclass of the type parameter's type T (such object is called a "companion object" of type T).

These criteria ensure a certain degree of locality of implicit arguments. For instance, clients can tailor the set of available arguments by selectively importing objects which define the arguments they want to see passed to implicit parameters.

If there are several eligible arguments which match the implicit parameter's type, the Scala compiler will chose a most specific one, using the standard rules of static overloading resolution. For instance, assume the call

```
sum(List(1, 2, 3))
```

in a context where stringMonoid and intMonoid are visible. We know that the formal type parameter a of sum needs to be instantiated to Int. The only eligible value which matches the implicit formal parameter type Monoid[Int] is intMonoid so this object will be passed as implicit parameter.

This discussion also shows that implicit parameters are inferred after any type arguments are inferred.

Implicit methods can themselves have implicit parameters. An example is the following method from module scala.List, which injects lists into the scala.Ordered class, provided the element type of the list is also convertible to this type.

```
implicit def list2ordered[a](x: List[a])
  (implicit elem2ordered: a => Ordered[a])
  : Ordered[List[a]] =
  ...
```

Assume in addition a method

```
implicit def int2ordered(x: int): Ordered[int]
```

that injects integers into the Ordered class. We can now define a sort method over ordered lists:

```
sort(xs: List[a])
    (implicit a2ord: a => Ordered[a]) = ...
```

We can apply sort to a list of lists of integers yss: List[List[int]] as follows:

```
sort(yss)
```

The Scala compiler will complete the call above by passing two nested implicit arguments:

```
sort(yss)(xs: List[int] => list2ordered[int](xs)(int2ordered)) .
```

The possibility of passing implicit arguments to implicit arguments raises the possibility of an infinite recursion. For instance, one might try to define the following method, which injects *every* type into the Ordered class:

```
def magic[a](x: a)(implicit a2ordered: a => Ordered[a])
  : Ordered[a] = a2ordered(x)
```

This function is of course too good to be true. Indeed, if one tried to apply sort to an argument arg of a type that did not have another injection into the Ordered class, one would obtain an infinite expansion:

```
sort(arg)(x => magic(x)(x => magic(x)(x => ... )))
```

To prevent such infinite expansions, we require that every implicit method definition is contractive. Here a method definition is *contractive* if the type of every implicit parameter type is "properly contained" [35] in the type that is obtained by removing all implicit parameters from the method type and converting the rest to a function type.

For instance, the type of `list2ordered` is

```
(List[a])(implicit a => Ordered[a]): Ordered[List[a]] .
```

This type is contractive, because the type of the implicit parameter, `a => Ordered[a]`, is properly contained in the function type of the method without implicit parameters, `List[a] => Ordered[List[a]]`.

The type of `magic` is

```
(a)(implicit a => Ordered[a]): Ordered[a] .
```

This type is not contractive, because the type of the implicit parameter, `a => Ordered[a]`, is the same as the function type of the method without implicit parameters.

Views

Views are implicit conversions between types. They are typically defined to add some new functionality to a preexisting type. For instance, assume the following trait if simple generic sets:

```
trait Set[T] {
  def include(x: T): Set[T];
  def contains(x: T): boolean
}
```

A view from class `GenList` to class `Set` is introduced by the following method definition.

```
implicit def listToSet[T](xs: GenList[T]): Set[T] =
  new Set[T] {
    def include(x: T): Set[T] =
      x prepend xs;
    def contains(x: T): boolean =
      !isEmpty && (xs.head == x || xs.tail contains x)
  }
```

Hence, if `xs` is a `GenList[`T`]`, then `listToSet(xs)` would return a `Set[`T`]`.

The only difference with respect to a normal method definition is the `implicit` modifier. This modifier makes views candidate arguments for implicit parameters, and also causes them to be inserted automatically as implicit conversions.

Say e is an expression of type T. A view is implicitly applied to e in one of two possible situations: when the expected type of e is not (a supertype of) T, or when a member selected from e is not a member of T. For instance, assume a value `xs` of type `List[T]` which is used in the following two lines.

```
val s: Set[T] = xs;
xs contains x
```

The compiler would insert applications of the view defined above into these lines as follows:

```
val s: Set[T] = listToSet(xs);
listToSet(xs) contains x
```

Which views are available for insertion? Scala uses the same rules as for arguments to implicit parameters. A view is available if it can be accessed without a prefix or it is defined in a companion object of either the source or the target type of the conversion. An available view is *applicable* if can be applied to the expression and it maps to the desired type, or to any type containing the desired member, in the case of a selection. Among all applicable candidates, Scala picks the most specific view. Here, specificity is interpreted in the same way as for overloading resolution in Java and Scala. It is an error if no view is applicable, or among the applicable views no most specific one exists.

Views are used frequently in the Scala library to upgrade Java's types to support new Scala traits. An example is Scala's trait `Ordered` which defines a set of comparison operations. Views from all basic types as well as class `String` to this type are defined in a module `scala.Predef`. Since the members of this module are imported implicitly into every Scala program, the views are always available. From a user's perspective, it is almost as if the Java classes are augmented by the new traits.

View Bounds

As presented so far, view methods have to be visible statically at the point of their insertion. Views become even more useful if one can abstract over the concrete view method to be inserted. This can be expressed by making the view an implicit parameter. An example is the following generic `maximum` method, which returns the maximum element of a non-empty list.

```
def maximum[T](xs: List[T])
    (implicit t2ordered: T => Ordered[T]): unit = {
  var mx = xs.head;
  for (val x <- xs.tail) if (mx < x) mx = x
  mx
}
```

This `maximum` function can be applied to any argument of type `List[T]`, where `T` is viewable as `Ordered[T]`. In particular, we can apply `maximum` to lists of basic types for which standard `Ordered` views exist.

Note that `maximum` uses a comparison operation `(mx < x)` on values `mx` and `x` of type `T`. The type parameter `T` does not have a comparison operation `<`, but there is the implicit parameter `t2ordered` which maps `T` into a type which does. Therefore, the comparison operation is rewritten to `(t2ordered(mx) < x)`.

The situation of associating a generic parameter with implicit views is so common that Scala has special syntax for it. A *view bounded* type parameter such as `[T <% U]` expresses that `T` must come equipped with a view that maps its values into values of type `U`. Using view bounds, the `maximum` function above can be more concisely written as follows:

```
def maximum[T <% Ordered[T]](xs: List[T]): unit = ...
```

This code is expanded into precisely the previous code for `maximum`.

10 Related Work

Scala's design is influenced by many different languages and research papers. The following enumeration of related work lists the main design influences.

Of course, Scala adopts a large part of the concepts and syntactic conventions of Java [23] and C# [15]. Scala's way to express properties is loosely modelled after Sather [44]. From Smalltalk [22] comes the concept of a uniform object model. From Beta [30] comes the idea that everything should be nestable, including classes. Scala's design of mixins comes from object-oriented linear mixins [6], but defines mixin composition in a symmetric way, similar to what is found in mixin modules [14, 25, 49] or traits [42]. Scala's abstract types have close resemblances to abstract types of signatures in the module systems of ML [24] and OCaml [29], generalizing them to a context of first-class components. For-comprehensions are based on Haskell's monad comprehensions [46], even though their syntax more closely resembles XQuery [3]. Views have been influenced by Haskell's type classes [47]. They can be seen as an object-oriented version of parametric type classes [38], but they are more general in that instance declarations can be local and are scoped. Classboxes [2] provide the key benefits of views in a dynamically typed system. Unlike views, they also permit local rebinding so that class extensions can be selected using dynamic dispatch.

In a sense, Scala represents a continuation of the work on Pizza [37]. Like Pizza, Scala compiles to the JVM, adding higher-order functions, generics and pattern matching, constructs which have been originally developed in the functional programming community. Whereas Pizza is backwards compatible with Java, Scala's aim is only to be interoperable, leaving more degrees of freedom in its design.

Scala's aim to provide advanced constructs for the abstraction and composition of components is shared by several recent research efforts. Abstract types are a more conservative construction to get most (but not all) of the benefits of virtual classes in gbeta [16, 17]. Closely related are also the delegation layers in FamilyJ [40] and work on nested inheritance for Java [32]. Jiazzi [31] is an extension to Java that adds a module mechanism based on *units*, a powerful form of parametrized module. Jiazzi supports extensibility idioms similar to Scala, such as the ability to implement mixins.

The Nice programming language [4] is a recent object-oriented language that is similar to Java, but has its heritage in $ML_{\leq}$ [5]. Nice includes multiple dispatch, open classes, and a restricted form of retroactive abstraction based on abstract interfaces. Nice does not support modular implementation-side typechecking. While Nice and Scala are languages which differ significantly from Java, they both are designed to interoperate with Java programs and libraries, and their compiler targets the JVM.

MultiJava [13] is a conservative extension of Java that adds symmetric multiple dispatch and open classes. It provides alternative solutions to many of the problems that Scala also addresses. For instance, multiple dispatch provides a solution to the binary method problem, which is addressed by abstract types in Scala. Open classes provide a solution to the external extensibility problem, which is solved by views in Scala. A feature only found in MultiJava is the possibility to dynamically add new methods to a class, since open classes are integrated with Java's regular dynamic loading process. Conversely, only Scala allows to delimit the scope of an external class extension in a program.

OCaml and Moby[20] are two alternative designs that combine functional and object-oriented programming using static typing. Unlike Scala, these two languages start with a rich functional language including a sophisticated module system and then build on these a comparatively lightweight mechanism for classes.

11 Conclusion

Scala is both a large and a reasonably small language. It is a large language in the sense that it has a rich syntax and type system, combining concepts from object-oriented programming and functional programming. Hence, there are new constructs to be learned for users coming from either language community. Much of Scala's diversity is also caused by the motivation to stay close to conventional languages such as Java and C#, with the aim to ease adoption of Scala by users of these languages.

Scala is also a reasonably small language, in the sense that it builds on a modest set of very general concepts. Many source level constructs are syntactic sugar, which can be removed by encodings. Generalizations such as the uniform object model allow one to abstract from many different primitive types and operations, delegating them to constructs in the Scala library.

Scala provides a powerful set of constructions for composing, abstracting, and adapting components. The aim is that with this set of components the language becomes extensible enough so that users can model their domains naturally in libraries and frameworks. Hence, there is less pressure to extend the language, because most constructions can be modeled conveniently in libraries. Examples of this abound already in the Scala libraries and applications. There are classes to model Erlang style actors, arbitrary precision integers, Horn clauses and constraints, to name just three. All of these constructions can be written as naturally as in a specialized language, yet integrate seamlessly into Scala (and by extension, Java).

This approach to extensibility transfers to some degree responsibility from language designers to users – it is still as easy to design a bad libraries as it is to design a bad language. But we hope that Scala's constructions will make it easier to design good libraries than existing mainstream languages.

Acknowledgments Scala's design and implementation was partially supported by grants from the Swiss National Fund under project NFS 21-61825, the Swiss National Competence Center for Research MICS, the European Framework 6 project PalCom, Microsoft Research, and the Hasler Foundation. Gilad Bracha, Craig Chambers, Erik Ernst, Matthias Felleisen, Shriram Krishnamurti, Gary Leavens, Sebastian Maneth, Erik Mejer, Oscar Nierstrasz, Klaus Ostermann, Didier Rémy, Mads Torgersen, and Philip Wadler have shaped the design of the language through lively and inspiring discussions. The contributors to the Scala mailing list have also given very useful feedback that helped us improve the language and its tools.

References

[1] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington. A monotonic superclass linearization for Dylan. In *Proc. OOPSLA*, pages 69–82. ACM Press, Oct 1996.

[2] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A Minimal Module Model Supporting Local Rebinding. In *Proc. JMLC 2003*, volume 2789 of *Springer LNCS*, pages 122–131, 2003.

[3] S. Boag, D. Chamberlin, M. F. Fermandez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML Query Language. W3c recommendation, World Wide Web Consortium, November 2003. `http://www.w3.org/TR/xquery/`.

[4] D. Bonniot and B. Keller. The Nice's user's manual, 2003. `http://nice.sourceforge.net/NiceManual.pdf`.

[5] F. Bourdoncle and S. Merz. Type-checking Higher-Order Polymorphic Multi-Methods. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–17, Paris, France, 1997.

[6] G. Bracha and W. Cook. Mixin-Based Inheritance. In N. Meyrowitz, editor, *Proceedings of* ECOOP '90, pages 303–311, Ottawa, Canada, October 1990. ACM Press.

[7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, eds. Extensible Markup Language (XML) 1.0. W3C recommendation, World Wide Web Consortium, February 2004. Available online `http://www.w3.org/TR/REC-xml-20040204/`.

[8] K. B. Bruce, M. Odersky, and P. Wadler. A Statically Safe Alternative to Virtual Types. *Lecture Notes in Computer Science*, 1445, 1998. Proc. ESOP 1998.

[9] K. B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language. In *Proceedings of* ECOOP '95, LNCS 952, pages 27–51, Aarhus, Denmark, August 1995. Springer-Verlag.

[10] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-Bounded Quantification for Object-Oriented Programming. In *Proc. of 4th Int. Conf. on Functional Programming and Computer Architecture, FPCA'89, London*, pages 273–280, New York, Sep 1989. ACM Pres.

[11] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An Extension of System F with Subtyping. *Information and Computation*, 109(1–2):4–56, 1994.

[12] K. Chen, P. Hudak, and M. Odersky. Parametric Type Classes. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 170–181, June 1992.

[13] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Design Rationale, Compiler Implementation, and User Experience. Technical Report 04-01, Iowa State University, Dept. of Computer Science, Jan 2004.

[14] D. Duggan. Mixin modules. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, volume 31(6), pages 262–273, 1996.

[15] ECMA. C# Language Specification. Technical Report Standard ECMA-334, 2nd Edition, European Computer Manufacturers Association, December 2002.

[16] E. Ernst. Family polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 303–326, Budapest, Hungary, 2001.

[17] E. Ernst. Higher-Order Hierarchies. In L. Cardelli, editor, *Proceedings ECOOP 2003*, LNCS 2743, pages 303–329, Heidelberg, Germany, July 2003. Springer-Verlag.

[18] M. O. et.al. An introduction to Scala. Technical report, EPFL Lausanne, Switzerland, Mar. 2006. Available online `http://scala.epfl.ch`.

[19] D. C. Fallside, editor. XML Schema. W3C recommendation, World Wide Web Consortium, May 2001. Available online `http://www.w3.org/TR/xmlschema-0/`.

[20] K. Fisher and J. H. Reppy. The Design of a Class Mechanism for Moby. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 37–49, 1999.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Massachusetts, 1994.

[22] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[23] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Java Series, Sun Microsystems, second edition, 2000.

[24] R. Harper and M. Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, January 1994.

[25] T. Hirschowitz and X. Leroy. Mixin Modules in a Call-by-Value Setting. In *European Symposium on Programming*, pages 6–20, 2002.

[26] A. Igarashi and M. Viroli. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. In *Proceedings of the Sixteenth European Conference on Object-Oriented Programming (ECOOP2002)*, pages 441–469, June 2002.

[27] M. P. Jones. Using parameterized signatures to express modular structure. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 68–78. ACM Press, 1996.

[28] R. Keller and U. Hölzle. Binary Component Adaptation. In *Proceedings ECOOP*, Springer LNCS 1445, pages 307–329, 1998.

[29] X. Leroy. Manifest Types, Modules and Separate Compilation. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 109–122, January 1994.

[30] O. L. Madsen and B. Moeller-Pedersen. Virtual Classes - A Powerful Mechanism for Object-Oriented Programming. In *Proc. OOPSLA'89*, pages 397–406, October 1989.

[31] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age Components for Old-Fashioned Java. In *Proc. of OOPSLA*, October 2001.

[32] N. Nystrom, S. Chong, and A. Myers. Scalable Extensibility via Nested Inheritance. In *Proc. OOPSLA*, Oct 2004.

[33] Oasis. RELAX NG. See `http://www.oasis-open.org/`.

[34] M. Odersky. Scala by example. Technical report, EPFL Lausanne, Switzerland, Mar. 2006. Available online `http://scala.epfl.ch`.

[35] M. Odersky. The Scala Language Specification. Technical report, EPFL Lausanne, Switzerland, Mar. 2006. Available online `http://scala.epfl.ch`.

[36] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS 2743, jul 2003.

[37] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 146–159, January 1997.

[38] M. Odersky, P. Wadler, and M. Wehr. A Second Look at Overloading. In *Proc. ACM Conf. on Functional Programming and Computer Architecture*, pages 135–146, June 1995.

[39] M. Odersky, C. Zenger, and M. Zenger. Colored Local Type Inference. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, pages 41–53, London, UK, January 2001.

[40] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, Malaga, Spain, 2002.

[41] B. C. Pierce and D. N. Turner. Local Type Inference. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, pages 252–265, New York, NY, 1998.

[42] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable Units of Behavior. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, Darmstadt, Germany, June 2003.

[43] J. Siek and A. Lumsdaine. Essential Language Support for generic programming. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, pages 73–84, Jun 2005.

[44] D. Stoutamire and S. M. Omohundro. The Sather 1.0 Specification. Technical Report TR-95-057, International Computer Science Institute, Berkeley, 1995.

[45] M. Torgersen, C. P. Hansen, E. Ernst, P. vod der Ahé, G. Bracha, and N. Gafter. Adding Wildcards to the Java Programming Language. In *Proceedings SAC 2004*, Nicosia, Cyprus, March 2004.

[46] P. Wadler. The Essence of Functional Programming. In *Proc.19th ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1992.

[47] P. Wadler and S. Blott. How to make *ad-hoc* Polymorphism less *ad-hoc*. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, January 1989.

[48] M. Zenger. Type-Safe Prototype-Based Component Evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.

[49] M. Zenger. *Programming Language Abstractions for Extensible Software Components*. PhD thesis, Department of Computer Science, EPFL, Lausanne, March 2004.