An Overview of the Scala Programming Language (2. Edition)

Philippe Altherr
8 PUBLICATIONS **543** CITATIONS
SEE PROFILE

Vincent Cremet
15 PUBLICATIONS **778** CITATIONS
SEE PROFILE

Gilles Dubochet
Ideas Belong
12 PUBLICATIONS **66** CITATIONS
SEE PROFILE

Burak Emir
Google Inc.
15 PUBLICATIONS **672** CITATIONS
SEE PROFILE

An Overview of the Scala Programming Language

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger

École Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland

Technical Report IC/2004/64

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 between 2001 and 2004 in the programming methods laboratory at EPFL. It 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 [35] to a concrete language design. Second, *symmetric 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).
- 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 symmetric 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).

```
// 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));
  }
}
```

Listing 1: A simple program in Java and Scala.

- Taken together, these constructs make it easy to express autonomous components using Scala libraries without need for special language constructs (Section 9).
- Scala allows external extensions of components using views (Section 10).
- Scala is currently implemented on the Java (Section 11) and .NET (Section 12) platforms.

Section 13 discusses related work and Section 14 concludes.

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 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[1]. This is why array accesses are written like function applications $a(i)$, instead of Java's $a[i]$.
- 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 1.5 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.

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 3.1 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.

[1]Arrays are further explained in Section 4.3.

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. standard coercions, see Section 10) 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 a standard coercion 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[2]:

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

At the bottom of the type hierarchy are the two classes scala.AllRef and scala.All. Type AllRef is a subtype of all reference types; its only instance is the **null** reference. Since AllRef 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 All is a subtype of every other type; there exist no instances of this type. Even though type All is empty, it is nevertheless useful as a type parameter. For instance, the Scala library defines a value Nil of type List[All]. 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 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 a *trait* specifying the interface of natural numbers. For now, traits can be seen as abstract classes; details are given later in Section 6.2. According to the definition of trait Nat, natural numbers provide two abstract methods isZero, and pred, as well as three concrete methods succ, +, and -.

```
trait 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 trait 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");
}
class Succ(n: Nat) extends Nat {
  def isZero: boolean = false;
  def pred: Nat = 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 [34]).

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 deter-

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

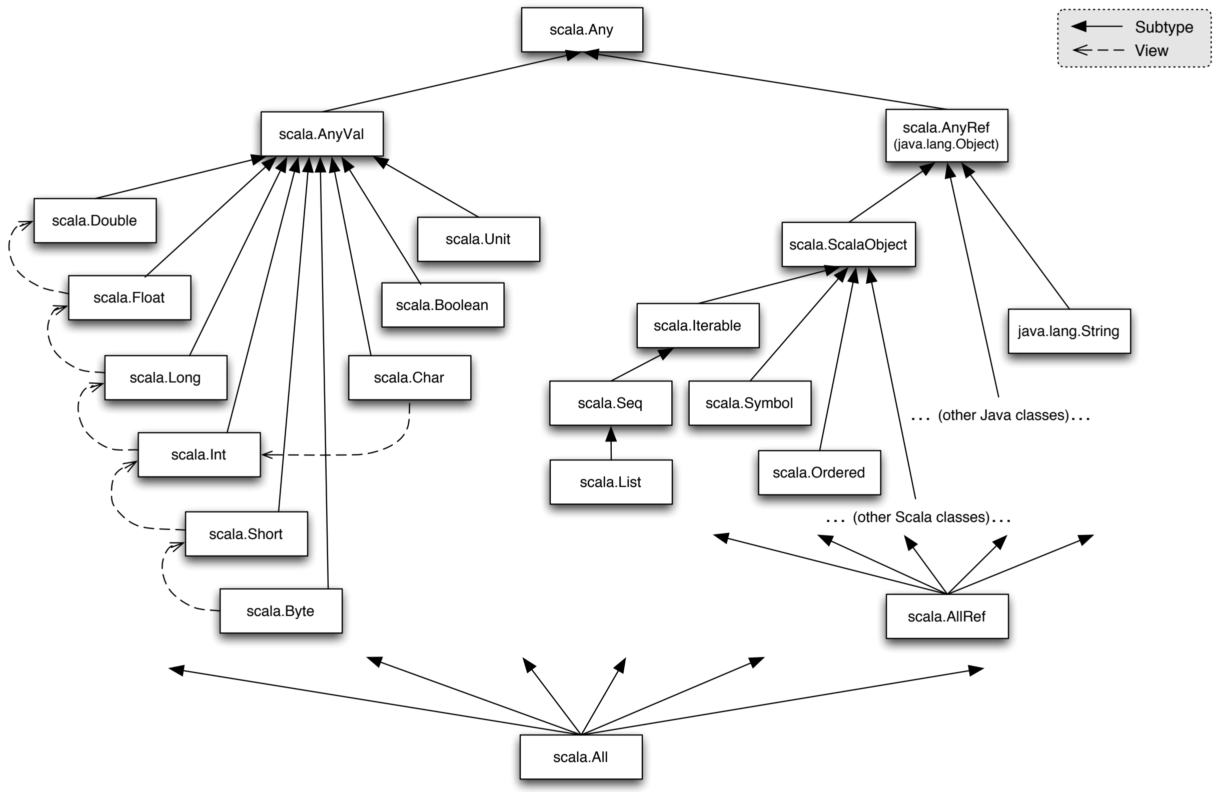

Figure 1: Class hierarchy of Scala.

mined 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 trait `Bool` that mimics the pre-defined booleans.

```
trait Bool {
  def && (def x: Bool): Bool;
  def || (def x: Bool): Bool;
}
```

In this trait, the formal parameter of methods `||` and `&&` is prefixed by **def**. The 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 && (def x: Bool): Bool = this;
  def || (def x: Bool): Bool = x;
}
object True extends Bool {
  def && (def x: Bool): Bool = x;
  def || (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.

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 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;
trait 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] {
  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 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[3], 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 1.5 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.

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

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)
}
```

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 [40, 38]. 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 of this trait 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 following trait `GenList` defines a simple covariant list with methods `isEmpty`, `head`, and `tail`.

```
trait 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[All] {
  def isEmpty: boolean = true;
  def head: All = throw new Error("Empty.head");
  def tail: List[All] = 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[All]` 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` trait. The most natural definition of this method takes an argument of the list element type:

```
trait 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 trait `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:

```
trait 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 trait `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.

```
trait 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 1.5 also has a way to annotate variances which is based on wildcards [43]. The scheme is essentially a syntactic variant of Igarashi and Viroli's variant parametric types [26]. Unlike in Scala, in Java 1.5 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 [35].

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.

```
trait SubjectObserver {
  type S <: Subject;
  type O <: Observer;
  class Subject: 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 trait `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 type annotation:

```
class Subject: 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.

Self-types were first introduced in the νObj calculus. They are relatively infrequently used in Scala programs, but they are nevertheless essential in situations where family polymorphism is combined with explicit references to **this**.

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 of 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

6.1 Class Reuse

The reuse of existing software components for the construction of new systems has many advantages: one can expect lower development costs due to a reduced development time, decreased maintenance requirements, as well as increased reliability and consistency.

Therefore, object-oriented programming languages are equipped with mechanisms that facilitate the reuse of existing software artifacts, like classes. This section presents and motivates Scala's class reuse mechanisms based on the following example program. This program defines a generic class `Buffer[T]` for assembling sequences of elements:

```
class Buffer[T] {
  var xs: List[T] = Nil;
  def add(elem: T): Unit = xs = elem :: xs;
  def elements: Iterator[T] = new BufferIterator;
  class BufferIterator extends Iterator[T] {
    var ys = xs;
    def hasNext: Boolean = !ys.isEmpty;
    def next: T = {
      val res = ys.head; ys = ys.tail; res
    }
  }
}
```

The implementation of class `Buffer` relies on the following iterator abstraction:

```
trait Iterator[T] {
  def hasNext: Boolean;
  def next: T;
}
```

Inheritance Like most mainstream object-oriented languages, Scala's primary class reuse mechanism is based on single inheritance; i.e., programmers can specialize classes by subclassing. To enrich class `Buffer` with additional methods `forall` and `exists`, we could, for instance, create a subclass of `IterableBuffer` defining the new functionality:

```
class IterableBuffer[T] extends Buffer[T] {
  def forall(p: T => Boolean): Boolean = {
    val it = elements; var res = true;
    while (res && it.hasNext) { res = p(it.next) }
    res
  }
  def exists(p: T => Boolean): Boolean = {
    val it = elements; var res = false;
    while (!res && it.hasNext) { res = p(it.next) }
    res
  }
}
```

Mixin-class composition The problem with the code above is its limited potential for reuse. Imagine there is an independent extension of class `Buffer` which models stacks:

```
class Stack[T] extends Buffer[T] {
  def push(elem: T): Unit = add(elem);
  def pop: T = { val y = xs.head; xs = xs.tail; y }
}
```

With single inheritance, it is impossible to reuse the existing definitions of the `forall` and the `exists` methods together with stacks. 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, in the definition of a new class. This mechanism makes it possible to combine `IterableBuffer` with `Stack`:

```
class IterableStack[T] extends Stack[T]
                       with IterableBuffer[T];
```

This program defines a class `IterableStack[T]` which inherits all definitions from `Stack[T]` and additionally includes the new definitions of `IterableBuffer[T]`. Mixing a class C into another class D is legal only as long as D's superclass is a subclass of C's superclass. Thus, the mixin composition in the program above is well-formed, since the superclass of `IterableStack` is a subclass of the superclass of `IterableBuffer`.

Scala enforces this requirement for type-safety reasons. Since only the delta of a class is "copied" into another class by a mixin-class composition, it could otherwise happen that some mixed-in members refer to inherited members which are not present in the new context, yielding a "method not found" exception.

Ambiguities In Scala, every class inherits exactly from one superclass and acquires class members from multiple other classes via mixin-based class composition. Imagine for instance the following subclass of `Buffer` which introduces a method `sameElements` together with an internally used `forall` method.

```
class ComparableBuffer[T] extends Buffer[T] {
  def forall(p: T => Boolean): Boolean = {
    val it = elements; var res = true;
    while (res && it.hasNext) { res = p(it.next) }
    res
  }
  def sameElements(b: IterableBuffer[T]): Boolean =
    forall(elem => b.exists(elem.equals));
}
```

We could derive a stack class `MyStack` offering functionality provided by both `IterableBuffer` and `ComparableBuffer` by using the two classes as mixins:

```
class MyStack[T] extends Stack[T]
                 with IterableBuffer[T]
                 with ComparableBuffer[T]; // error!
```

In Scala, methods defined in mixins either represent new methods or they override the respective methods in the actual superclass. As the previous example shows, it may happen that two mixins define the same method. For class `MyStack` it is unclear which `forall` method to use. This ambiguity constitutes a compile-time error which has to be resolved by the programmer explicitly. A possible solution is to introduce a new `forall` method which forwards the call to the desired implementation. The following program makes use of the **super**`[C]` primitive, which allows one to refer to concrete definitions in the mixin class `C`:

```
class MyStack[T] extends Stack[T]
                 with IterableBuffer[T]
                 with ComparableBuffer[T] {
  override def forall(p: T => Boolean) =
    super[IterableBuffer].forall(p);
}
```

6.2 Traits

Besides ambiguities, another serious problem of multiple inheritance is the *diamond inheritance* dilemma which appears if a class inherits from two other classes that share superclasses. Without further restrictions, these superclasses would get inherited twice in this scenario. This would lead to a duplication of the state encapsulated by these superclasses and therefore would result in serious consistency issues.

To avoid this, Scala allows a class to be mixed into another class only if it has not been used before in the other class as either superclass or mixin. Unfortunately, this rule is very restrictive, ruling out many cases where inheriting twice from the same class would not constitute a problem — this is the case in particular for classes that do not encapsulate state (interfaces in Java fall into that category). For this reason, Scala introduces the notion of *traits*. Traits are abstract classes that do not encapsulate state, neither in form of variable definitions nor by providing a constructor with parameters. Opposed to interfaces in Java though, they may implement concrete methods.

Since traits do not encapsulate state, inheriting twice from a trait is legal in Scala. It is therefore possible to have the same trait multiple times in the superclass hierarchy of a class.

Reuse of class `IterableBuffer` is also restricted by the requirement that it can only be mixed into classes that subclass `Buffer`. But the functionality provided by `IterableBuffer` only depends on the existence of an `elements` method. Scala makes it possible to express this in the following form:

```
trait Iterable[T] {
  def elements: Iterator[T];
  def forall(p: T => Boolean): Boolean = {
    val it = elements; var res = true;
    while (res && it.hasNext) { res = p(it.next) }
    res
  }
  def exists(p: T => Boolean): Boolean = {
    val it = elements; var res = false;
    while (!res && it.hasNext) { res = p(it.next) }
    res
  }
}
```

Trait `Iterable` defines methods `forall` and `exists` as before, but *defers* the definition of method `elements` — it is *abstract* in the terminology of Java. As opposed to class `IterableBuffer`, this trait can be mixed into all classes. If `Iterable` is mixed into a class without a concrete `elements` method, then the resulting class will have a deferred `elements` method, otherwise, the concrete method will implement the deferred method mentioned in trait `Iterable`. Thus, concrete methods always override abstract ones in mixin-class compositions. This principle is exploited in the following alternative definition of class `IterableBuffer`:

```
class IterableBuffer[T] extends Buffer[T]
                        with Iterable[T];
```

6.3 Layering Classes and Traits

Scala's mixin-class composition mechanism makes it easy to compose complex classes with extensive functionality from smaller and simpler ones that model only particular aspects. The previous section showed how traits can be used to create generic abstractions that can be used to add new methods or to implement deferred methods of existing classes. Traits mixed into a class can also override existing methods, as the following code fragment shows:

```
trait Displayable[T] {
  def elements: Iterator[T];
  override def toString(): String = {
    val res = new StringBuffer;
    for (val elem <- elements) res.append("␣" + elem);
    res.toString()
  }
}
class DisplayableBuffer[T] extends IterableBuffer[T]
                           with Displayable[T];
```

Class `DisplayableBuffer[T]` now provides the `toString()` method defined in trait `Displayable[T]`.

The presented technique for overriding methods only works if the method in the superclass or in the mixed in traits is concrete. But often, one would like to define generic traits that enhance an existing method by overriding without that method being concrete in the superclass. As an example, consider providing a building block for making iterators synchronized such that they can be used in a concurrent setting. A naive definition would be:

```
// erroneous trait definition
trait SynchronizedIterator[T] extends Iterator[T] {
  override def next: T = synchronized { super.next }
  override def hasNext: Boolean =
    synchronized { super.hasNext }
}
```

This definition is illegal, because supertrait `Iterator` does not provide concrete implementations for both methods `next` and `hasNext`. Thus, the **`super`** reference is not allowed.

Scala still allows programmers to define such an abstraction. However, it requires that an **`abstract`** modifier is used for all those methods which override abstract methods in the static superclass, yet which are supposed to override concrete methods in a mixin composition.

```
trait SynchronizedIterator[T] extends Iterator[T] {
  abstract override def next: T =
    synchronized { super.next }
  abstract override def hasNext: Boolean =
    synchronized { super.hasNext }
}
```

Classes containing methods tagged with both **`abstract`** and **`override`** cannot be instantiated — they have to be declared **`abstract`** themselves. Furthermore, such classes can only be mixed into classes that provide concrete versions for all mixed in methods flagged with **`abstract`** and **`override`**.

Traits like `SynchronizedIterator` are extremely useful for synchronizing arbitrary iterator implementations simply by a mixin composition. For instance, we could implement a synchronized iterator for the `Buffer` class based on an unsynchronized version as defined by class `BufferIterator` at the beginning of Section 6.1.

```
class Buffer[T] {
  ...
  def elements: Iterator[T] =
    new BufferIterator with SynchronizedIterator[T];
}
```

In mainstream object-oriented languages like Java, programmers would typically separate concerns using *object composition* techniques. Here, a generic synchronized iterator class would provide synchronized methods that forward the call to the corresponding methods of another unsynchronized iterator. The advantage of this approach is its flexibility, since it can be decided dynamically whether to use a synchronized or unsynchronized iterator. On the other hand, this approach does not guarantee statically that the synchronization scheme is adhered to at runtime, since programmers can possibly circumvent the programming pattern easily by exposing the unsynchronized iterator.

The approach based on mixin *class composition* turns iterators statically into synchronized iterators guaranteeing that synchronization cannot be broken by programmers at runtime. In Scala, the programmer has the choice between composing abstractions at runtime using object composition or at compile-time using class composition. Whether one uses object or class composition depends predominantly on the particular flexibility and safety requirements.

6.4 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 [47].

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 [48].

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:

```
trait 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 trait `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 constructor 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:

```
trait 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 nested 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`.[4]

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 [18], 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.

[4]Patterns in Scala are linear in the sense that a variable may appear only once within a pattern.

8.3 Regular Sequence Pattern 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.nobinding.XML.load("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 regular sequence pattern _* matches an arbitrary sequence). Then it constructs a new phonebook element with child sequence cs followed by the node newEntry.

Regular sequence patterns extend conventional algebraic patterns discussed in Section 7 with the regular expression constructs * (zero to arbitrary repetition of a sequence), ? (zero or one occurrence of a sequence), and | (to describe an alternative of sequences). They can be applied to any sequence, i.e. any instance of Seq[A]. 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 search for the sequence of letters "Gogle" or "Google", or .... If the input z matches, then the function returns what remains after the occurrence, otherwise it generates a runtime error. Ambiguities that emerge (e.g., for several occurrences of 'Go*gle-words' in z) are resolved using the (left) shortest match policy which chooses the shortest match for each possibility (such as a *), coming from the left. In the example this coincides with matching the *first* occurrence of "Goo*gle" in the input z.

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 Autonomous Components

The Scala language as such does not provide any primitives for concurrent programming. Instead the core language has been designed to make it easy to build libraries to provide different concurrency models built on top of the thread model of the underlying host language. In this section we will exemplify the power of Scala by implementing a small library for fault-tolerant active objects with a flavor of Erlang-like actors [1].

Imagine for example that you have written a server class MyServer that is invoked by calling the method startServer. Scala makes it possible to make this server concurrent just by mixing it into the thread class.

```
class MyConcurrentServer extends Thread with MyServer {
  override def run() = startServer;
}
```

We can generalize the code above to a class, lets call it Process, which can take arbitrary code and execute it in a separate thread.

```
class Process(def body: unit) extends Thread {
  override def run() = body;
}
```

To make it even easier to spawn new processes we can implement a spawn function in an object.

```
object Process {
  def spawn(def body: unit): Process = {
    val p = new Process(body); p.start(); p
  }
}
```

Now we can start a process, but how do we stop it? Well a process stops when it has no more code to execute, i.e., when the code in body reaches its end. Sometimes we would like to kill the process prematurely, we can do this by adding an exit method to the Process class.

```
class Process(def body: unit) extends Thread {
  private var exitReason: AnyRef = null;
  override def run() = {
    try { body }
    catch {
      case e: InterruptedException =>
        exitReason.match {
          case null => Console.println(
            "Process␣exited␣abnormally␣" + e);
          case _ => Console.println(
            "Process␣exited␣with␣reason:␣" +
            exitReason);
        }
    }
  }
  def exit(reason: AnyRef): unit = {
    exitReason = reason; interrupt()
  }
}
```

Just running an object in a separate thread does not give us true active objects. All method calls and field accesses from other threads have to be synchronized in order to be safe. Adding this synchronization by hand is error prone and can easily lead to deadlocks or inefficient code.

A better approach to thread communication is to use message passing, e.g., by implementing an Erlang like actor model. In Erlang each actor (or process in the Erlang terminology) has a mailbox to which other processes can asynchronously send messages. The process owning the mailbox can selectively receive messages from the mailbox. In Scala we can implement mailboxes with the following signature.

```
class MailBox {
  def send(msg: Any): unit;
  def receive[a](f: PartialFunction[Any, a]): a;
  def receiveWithin[a](msec: long)
                     (f: PartialFunction[Any, a]): a;
}
```

Messages are added to the mailbox by the `send` method. Messages are removed using the `receive` method, which is passed a message processor `f` as argument, which is a partial function from messages to some arbitrary result type. Typically, this function is implemented as a pattern matching expression. The `receive` method blocks until there is a message in the mailbox for which its message processor is defined. The matching message is then removed from the mailbox and the blocked thread is restarted by applying the message processor to the message. Both sent messages and receivers are ordered in time. A receiver r is applied to a matching message m only if there is no other (message, receiver) pair which precedes (m, r) in the partial ordering on pairs that orders each component in time.

We can now extend our `Process` class by mixing in the `MailBox` class.

```
class Process(def body: unit) extends Thread
                              with MailBox {
  ...
}
```

In order to build fault-tolerant systems it is imperative that we can detect failures in a process. This can be achieved by making it possible to *link* processes. When a process (A) is linked to another process (B), A will send a signal to B when A dies. This makes it possible to monitor the failure of processes and to implement supervision trees where a supervisor process monitors worker processes and can restart them if they fail.

To implement this in Scala we have to add a list of links to the `Process` class and provide the link methods, as well as signal a failure to all linked processes. We can now give the complete `Process` class, see Listing 2.

We can use the `Process` class to implement a small counter server (see Listing 3). This server implements a counter that can be incremented and read by sending the messages `Increment`, and `GetValue` respectively. The server itself consists of only one method, the `server` method. The object `Counter` provides a functional interface to the counter process.

```
class Signal extends Message;
case class Normal() extends Signal;
case class Exit(p: Process, m: Message)
           extends Message;

class Process(def body: unit) extends Thread
                              with MailBox {
  private var exitReason: AnyRef = null;
  private var links: List[Process] = Nil;
  override def run() =
  try { body; signal(Normal()) }
  catch {
    case _: InterruptedException =>
      signal(exitReason);
    case exitSignal =>
      signal(exitSignal);
  }
  private def signal(s: Message) =
    links.foreach(
      p: Process => p.send(Exit(this, s)));
  def !(msg: Message) = send(msg);
  def link(p: Process) = links = p::links;
  def unlink(p: Process) =
    links = links.remove(p2 => p == p2);
  def spawnLink(def body: unit) = {
    val p = new Process(body);
    p.link(this); p.start(); p
  }
  def self = this;
  def exit(reason: AnyRef): unit = {
    exitReason = reason; interrupt()
  }
}
```

Listing 2: The `Process` class.

10 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 [20]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].

```
object Counter {
  class Messages();
  case class Increment() extends Messages;
  case class GetValue(from: Process) extends Messages;
  case class Stop() extends Messages;
  case class Value(value: int) extends Messages;
  def start: Process = spawn(server(0));
  def increment(Counter: Process): unit =
    Counter ! Increment();
  def value(Counter:Process):Int = {
    Counter ! GetValue(self);
    receive { case Value(value) => value }
  }
  def stop(Counter: Process): unit = Counter ! Stop();
  private def server(v:int):unit = {
    var stop = false;
    var value = v;
    while(!stop) {
      receive {
        case Increment() => value = value+1;
        case GetValue(from) => from ! Value(value);
        case Stop => stop = true;
      }
    }
  }
}
```

Listing 3: Example of the use of processes, a simple server.

10.1 Views

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 follow some of the intuitions of Haskell's type classes, translating them into an object-oriented approach. Unlike with type classes, the scope of a view can be controlled, and competing views can coexist in different parts of one program.

A view is introduced by a normal Scala method definition which defines an entity named `view`. For instance, assume the following trait for 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.

```
def view[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 `view(xs)` would return a `Set[`T`]`.

The only difference with respect to a normal method definition is that views are inserted automatically by the Scala compiler. 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] = view(xs);
view(xs) contains x
```

Which views are available for insertion? Scala considers as candidates all views which can be accessed at the point of insertion without a prefix expression. This includes views defined locally or in some enclosing scope, as well as views inherited from base classes or imported from other objects by an **import** clause. Shadowing is not taken into account. That is, a local view does not hide a view defined in an enclosing scope. A 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). Among all candidates, Scala picks the most specific applicable 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.

Locality is ensured by the restriction that only those views accessible without a prefix are candidates. Clients can tailor the set of available views by selectively importing objects defining views.

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.

10.2 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. An example is the following generic `maximum` method, which returns the maximum element of a non-empty list.

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

The method has a *view bounded* type parameter `[T <% Ordered[T]]`. This type parameter can be instantiated to any type T which is a subtype of, or viewable as `Ordered[`T`]`. In particular, we can apply `maximum` to lists of basic types for which standard `Ordered` views exist.

Note that a view method application needs to be inserted in the `mx < x` condition in method `maximum`. Where does this view method come from? Since it is not statically known at the point of insertion, it must be passed as a parameter. In fact, for every view-bounded type parameter `[`t `<:` T`]`, an implicit value parameter `(view:` t `=>` T`)` is added to the parameter list of a class or method. When the class constructor or method is called, a concrete view method which matches the view parameter type is passed. The selection of this view method is analogous to the view selection for type conversions discussed in the last section.

For instance, the method call

```
maximum(List(1, -3, 42, 101))
```

would be completed to

```
maximum(view)(List(1, -3, 42, 101))
```

where `view` is the view method from `int` to `Ordered[int]` defined in `scala.Predef`.

10.3 Conditional Views

View methods might themselves have view-bounded type parameters. This allows the definition of conditional views. For instance, it makes sense to compare lists lexicographically as long as the list element type admits comparisons. This is expressed by the following view method:

```
def view[T <% Ordered[T]](x: List[T]) =
  new Ordered[List[T]] {
    def < (y: List[T]): boolean =
      !y.isEmpty &&
       (x.isEmpty || x.head < y.head ||
        x.head == y.head && x.tail < y.tail)
  }
```

The method maps elements of type `List[`T`]` to instances of type `Ordered[List[`T`]]` as defined in Section 5.1, provided the list element type T is itself viewable as `Ordered[`T`]`.

11 Implementation

The Scala compiler compiles Scala source code to JVM class files. It supports separate compilation by storing Scala type information in the generated class files as an attribute.

The compiler consists of several phases. The first one, the parser, reads all source files and generates an abstract syntax tree. This tree is then passed to the successive phases which annotate it or transform it. Finally, the tree is linearized and translated to JVM bytecode.

The next sections describe the different compiler phases and the transformations applied to the Scala code. For reasons of space we leave out some phases and transformations.

11.1 Parsing

The parser consists of a hand-written scanner and parser. The parser is a standard top-down parser. The scanner is the combination of two scanners: the Scala scanner, and the XML scanner which is needed to parse the XML literals.

11.2 Code Analysis

The code analysis phase is indisputably the most complex of all compiler phases. This is partly due to the number of tasks it performs, including

- name analysis
- overloading resolution
- view resolution
- type inference
- type analysis
- constraint checks.

However, most of the difficulty comes from the fact that these tasks have to be performed simultaneously because each tasks needs some information provided by one or several others. The name analysis, the type analysis and the overloading resolution are similar to what is done in Java compilers. The main difference is the presence of type parametrized classes and functions which often may be omitted and have to be inferred by the compiler. The type inference is based on the colored local type inference [38].

11.3 Functions with **def** parameters

Functions with **def** parameters are rewritten to functions where each **def** parameter of type T is replaced by a normal parameter of type `Function0[`T`]`. For example, the definition

```
def or(x: Boolean, def y: Boolean): Boolean =
  if (x) true else y;
```

is rewritten to

```
def or(x: Boolean, y: Function0[Boolean]): Boolean =
  if (x) true else y.apply();
```

The example shows also that references to **def** parameters are replaced by a call to the `apply` method of the rewritten parameters.

At a call site, arguments corresponding to **def** parameters are replaced by instances of `Function0`. For example, the expression `or(isFoo(), isBar())` is replaced by the following expression.

```
or(isFoo(), new Function0[Boolean] {
  def apply(): Boolean = isBar()
})
```

11.4 Curried Functions

The JVM supports only functions with one parameter section. Therefore functions with multiple parameter sections must be eliminated. This is done by merging all sections into one. For example, the function

```
def sum(x: Int)(y: Int): Int = x + y;
```

is replaced by the following one

```
def sum(x: Int, y: Int): Int = x + y;
```

Partial applications of functions with multiple parameter sections are replaced by anonymous functions. For example, the definition

```
val inc: Int => Int = sum(1);
```

is replaced by

```
val inc: Int => Int = y => sum(1)(y) .
```

11.5 Pattern Matching Expressions

Pattern matching expressions are translated into automata such that the number of tests needed to find the first matching pattern is minimized. Algebraic patterns as discussed in Section 7 and simple sequence patterns are translated with the technique used by the extensible Java compiler JaCo [49, 46]. The translation scheme corresponds closely to the one introduced by the Pizza compiler [36].

The more powerful regular expression patterns discussed in Section 8.3 use a different translation scheme which is based on the theory of regular tree grammars.

11.6 Local Classes and Functions

Both Scala and Java support local classes (i.e., classes defined in the body of a function) and Scala also supports local functions. These local definitions are eliminated by lifting them out into the next enclosing class; local classes become new private classes of the enclosing class and local functions become new private methods.

The main difficulty of lifting definitions is the possible presence of references to variables of the enclosing function in the body of the lifted definition. These references become invalid after the lifting because the referenced variables are no longer in the scope of the lifted definition. This is solved by adding to the lifted definition a new argument for every referenced variable of the enclosing function and by replacing each reference to one of these variables by a reference to the corresponding new argument.

This solution works well as long as all referenced variables are immutable. This is the case in Java but not in Scala. With mutable variables, the described solution fails because changes that occur after the lifted class is created or the lifted function is called will never be noticed. Furthermore, there is no way to modify the value of the variable from within the lifted definition.

To overcome these problems, mutable variables referenced by local definitions are first transformed into `Cells`. For example the variable definition

```
var i: Int = 0;
```

would be transformed into

```
val i: Cell[Int] = new Cell(0);
```

if it was referenced by a local definition.

11.7 Inner Classes

Although Java supports inner and nested classes, the JVM supports neither of them. Therefore Java compilers have to transform those classes into top-level classes. The Scala compiler uses techniques similar to those used by Java compilers.

Inner classes are transformed into nested classes by adding a new field containing a reference to the enclosing class instance and by replacing all references to this instance by references to this new field. All constructors are also augmented with a new argument needed to initialize the new field.

The transformation of nested classes into top-level classes involves giving them a non-conflicting top-level name. The only difficulty is the possible presence of references to private members of the enclosing class. These references would be illegal in the resulting top-level class. To avoid them, a new package-private access member is added to the enclosing class for each of its private members that is referenced by an nested class and all references in nested classes to these private members are replaced by references the corresponding access members.

11.8 Mixin Expansion

In Scala, every class may contain code and every class may be used as a mixin. Therefore, by using mixins, it is possible to define classes that inherit code from several other classes. In the following example

```
trait A {
  def foo: String;
  def bar: String;
}
class B extends A {
  def foo: String = "foo";
}
class M extends A {
  def bar: String = "bar";
}
class C extends B with M;
```

the class `C` inherits the implementation of method `foo` from class `B` and the implementation of method `bar` from class `M`.

The JVM supports only single class inheritance. Therefore, code inheritance from multiple classes has to be simulated. This is done by copying the code that can't be inherited.

In addition to class inheritance, the JVM supports multiple interface inheritance. An interface is an abstract class that only declares members but contains no code. This makes it possible to replicate any Scala type hierarchy on the JVM.

To do so, every class C is split into a trait C^T and a class C^C. The trait C^T contains a member declaration for every member declared in class C and, assuming that class C extends class S and mixins $M_0, ..., M_n$, it extends the traits S^T and $M_0^T, ..., M_n^T$. The class C^C extends the class S^C and the trait C^T and receives all the code from class C. In addition to that, all the code from the mixins $M_0, ..., M_n$ is duplicated in class C^C.

The code below shows how the example is transformed. Note that C^T keeps the name of C and that the name of C^C is obtainend by appending `$class` to the name C.

```
trait A {
  def foo: String;
  def bar: String;
}
abstract class A$class with A;
trait B with A {
  def foo: String;
}
class B$class extends A$class with B {
  def foo: String = "foo";
}
trait M with A {
  def bar: String;
}
class M$class extends A$class with M {
  def bar: String = "bar";
}
```

```
trait C with B with M;
class C$class extends B$class with C {
  def bar: String = "bar";
}
```

It can be seen that class M$class inherits the implementation of method foo from class B$class, but that the implementation of method bar in class M$class is not inherited and has to be duplicated in class C$class.

The trait C^T replaces all occurrences of class C in all types. This is possible because, by construction, it exhibits the same inheritance graph as class C and has the same members. Since, it contains and inherits no code, the trait C^T can be mapped to a JVM interface. The class C^C replaces all occurrences of class C in instance creations. It is mapped to a JVM class.

11.9 Type Mapping

At this point, after all the previously mentioned transformations, the mapping from Scala classes to JVM classes is straightforward: every Scala class is mapped to a JVM class with the same name. There are only two exceptions: classes Any and AnyRef which are both mapped to java.lang.Object.

For performance reasons, instances of subclasses of AnyVal are treated separately. These values are usually represented by their corresponding JVM primitive values and not by instances of their class. The subclasses of AnyVal are also replaced by their corresponding JVM primitive type in function and variable declarations. For example, the method **def** size: Int is compiled to a method with the JVM primitive type int as return type. Instances of subclasses of AnyVal are created only when values of these types are passed where a value of type Any or AnyVal is expected.

For performance reasons, instances of class Array are also treated separately; these values are usually represented by JVM native arrays. Instances of Array are created only exceptionally, for example when an array is passed where an instance of Array[T], with T <: Any, is expected because the JVM has no native array type that is a super-type of all other array types.

11.10 Type Erasure

Although Java 1.5 introduces type parameterized classes and methods to the language, the JVM still does not support this. Therefore, all type parameter sections are removed and the remaining type variables are replaced by their bounds. Thus, types like List[Int] become List and function definitions like

```
def id[T](x: T): T = x;
```

become

```
def id(x: Any): Any = x;
```

Sometimes, a type cast needs to be added. This happens when a function whose return type is a type variable is called or when a variable whose type is a type variable is used. For example, the expression id[String]("hello").length() has to be replaced by id("hello").asInstanceOf[String].length() because the function id, after transformation, has the return type Any.

This technique is also known as *type erasure*. It is used by several other compilers, for example in the GJ compiler. It has been formalized by Igarashi, Pierce, and Wadler in [25]. They also prove that all added type casts are safe (they never raise a ClassCastException).

11.11 Code Generation

In the end, the code is in a shape such that it can be easily linearized and converted to JVM class files. This is done in the last phase of the compiler. The class file generation relies on FJBG: a home-grown library for fast class file generation.

11.12 Implementation Language

In the beginning, the whole compiler was written in Java, or more precisely in Pico, a Java dialect with algebraic data types. Since then some phases, including the scanner, the parser and the analyzer have been rewritten in Scala. Our goal is to have a compiler entirely written in Scala. Most of the new code is now directly written in Scala and the old Java code is slowly rewritten in Scala.

The library is almost entirely written in Scala. The only exceptions are some internal runtime classes and the value type classes along with the class Array which require some special handling from the compiler as described in Section 11.9.

12 Scala for .NET

The .NET platform is built around the Common Language Infrastructure (CLI) [15] which provides a specification for executable code and the execution environment in which it runs. The Common Language Specification (CLS) is a subset of the CLI that defines rules for language interoperability. While Scala has been developed mostly with focus on JVM, the aim is to support all CLS compliant features of .NET, i.e. to be a CLS *consumer*.

12.1 Class and Method Mappings

The differences between the JVM and .NET start with the root class on both platforms, namely java.lang.Object and System.Object. Scala abstracts over this difference and introduces the type scala.AnyRef as the root of the hierarchy for reference types (Section 3.1). In the .NET version of Scala, AnyRef is an alias for System.Object, rather than java.lang.Object.

The root class java.lang.Object defines several methods, among them the familiar equals, hashCode and toString methods. On .NET, the root class System.Object defines semantically equivalent methods but with different names. Since AnyRef is just an alias for one of the root classes, it can be expected that on .NET its methods will have names as defined in System.Object. However, this would fragment the Scala language and preclude the possibility to write Scala programs that compile and run on the two platforms without modifications.

The root of the Scala class hierarchy is scala.Any, which is the direct superclass of AnyRef (Figure 3.1). It already defines the equals, hashCode and toString methods. This necessitates a translation of the names of the equivalent methods of System.Object (Equals $\to$ equals, ToString $\to$ toString, GetHashCode $\to$ hashCode), so that they override the corresponding method in Any. This means that, say, the ToString method of any .NET type is accessible as toString from a Scala program. Furthermore, defining a toString method in a Scala class will effectively override System.Object.ToString. To avoid confusion, especially among .NET programmers, the compiler will reject any attempt to override any of these methods under its original System.Object name.

```
trait A {
  // ok, overrides System.Object.ToString
  override def toString() = "A";

  // compilation error, should be 'toString'
  override def ToString() = "A";

  // ok, just another method
  def ToString(prefix: String): String;
}
```

Sometimes translating the name of a method is not enough. The getClass method of java.lang.Object returns an instance of java.lang.Class, which is primarily used for reflection. System.Object defines a similar method, GetType, but it returns an instance of System.Type, which serves the same purposes as java.lang.Class. While it is possible to implement the getClass method for .NET (using the J# runtime libraries), this method is considered to be platform-specific and is not present in the .NET version of Scala. One should use the GetType method and the native .NET reflection facilities.

Another platform-specific feature is object cloning. On the JVM this is supported by the clone method of java.lang.Object. Every class that requires cloning has to implement the java.lang.Cloneable interface and override the clone method. On .NET, System.Object defines the MemberwiseClone method, which returns a field-by-field (or *shallow*) copy of an object and cannot be overridden. Object cloning is supported by implementing the System.ICloneable interface which declares a Clone method that has to be overridden in the implementing class.

```
class MyCloneable with ICloneable {
  def Clone() = super.MemberwiseClone();
}
```

To improve source-level compatibility, many additional methods of java.lang.String have to be mapped to the appropriate System.String methods. At the same time, all methods of System.String are accessible under their usual names.

```
// java.lang.String.substring(int, int)
val s1 = "0123".substring(1, 3)); // "12";

// System.String.Substring(int, int)
val s2 = "0123".Substring(1, 3)); // "123";
```

12.2 Properties

.NET properties are a metadata-level mechanism to associate getter and/or setter methods with a single common name. C# [14] introduces special syntax for definining properties.

```
public class Celsius {
  private int d = 0;
  public int degree {
    get { return d; }
    set { if (value >= -273) d = value; }
  }
}
```

The value of a property is obtained using its name (as with a field); to set a new value, the field assignment syntax is used.

```
Celsius c = new Celsius(); c.degree = c.degree - 1;
```

Scala employs a similar technique in its treatment of variables, which can be extended to defining properties. The getter and setter methods of a .NET property are translated according to the Scala convention (Section 3.3) and can be used as if they were defined in Scala.

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

Properties in .NET can have parameters (*indexed* properties). In C#, they are declared using special syntax.

```
abstract class Map {
  public abstract Object this[Object key] { get; set; }
}
```

To give access to such properties, C# employs an array indexing syntax.

```
public void inverse(Map map, Object key) {
  Object v = map[key];
  map[v] = key;
}
```

Such properties can be translated according to the scheme used to implement arrays in Scala (Section 4.3). The getter of an indexed property is renamed to apply, and the setter to update. Then, from a Scala perspecive, the class Map will look like this.

```
abstract class Map {
  def apply(key: Any): Any;
  def update(key: Any, value: Any): unit;
}
```

And can be used in a way, similar to C#.

```
def inverse(map: Map, key: Any) = {
  val v = map(key); map(v) = key;
}
```

12.3 Value Types

User-defined value types are a novel feature of the .NET framework. They obtain their special status by extending System.ValueType. Objects of value types are allocated on the stack, as opposed to reference types which are allocated on the heap. They are also passed by value when used as method arguments. In many situations, for instance generic collection libraries, the code is written to handle reference types. Fortunately, every value type can be represented as an instance of a reference type. The Scala compiler will statically determine the need for such conversion and generate the appropriate code.

Scala's notion of value types does not extend to user-definable value types. Consequently, one cannot define .NET value types in Scala. They have to be provided in an external .NET binary file called an *assembly*.

Structures. Value types are useful for representing lightweight objects. In C# they are called *structures* and are defined using special syntax.

```
struct Point {
  public Point(int x, int y) { this.x = x; this.y = y; }
  public int x, y;
}
```

Once defined in a .NET assembly, structures can be used in Scala programs like regular reference types.

```
def r(p: Point): double = Math.Sqrt(p.x*p.x + p.y*p.y);
def dist(p1: Point, p2.Point): double = {
  val p = new Point(p1.x - p2.x, p1.y - p2.y);
  r(p)
}
```

Enumerations. .NET has native support for type-safe enumerations. An enumeration type extends `System.Enum` and defines a value type, since `System.Enum` extends `System.ValueType`. The members of an enumeration are named constants of any integral type (`int`, `short`, etc.) except for `char`. Every enumeration defines a distinct type and its members cannot be used as values of the enumeration's underlying type; this is only possible with an explicit cast.

In C#, an enumeration is defined using special syntax.

```
public enum Color {
  Red, Green, Blue
}
```

When a .NET enumeration is used in a Scala program, it is treated as a reference type. Its members are seen as static fields that have the type of the enumeration.

```
class Color extends System.Enum;
object Color {
  val Red: Color;
  val Green: Color;
  val Blue: Color;
}
```

In a .NET assembly these fields are represented as *literals*. Literals have fixed values which reside in the assembly metadata, and they cannot be referenced at runtime. Instead, the compiler inlines the value associated with the field.

Every enumeration type is augmented with methods that perform comparisons (`==`, `!=`, `<`, `<=`, `>`, `>=`) and bitwise logical operations (`|`, `&`, `^`).

```
def isRed(c: Color): Boolean = (c == Color.Red);
```

However, such methods do not exist in the definition of the enumeration. They are recognized by the compiler and implemented to perform the respective primitive operation on the numerical values associated with the members of the enumeration.

13 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 [22] and C# [14]. Scala's way to express properties is loosely modelled after Sather [42]. From Smalltalk [21] 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 [13, 24, 48] or traits [41]. Scala's abstract types have close resemblances to abstract types of signatures in the module systems of ML [23] and OCaml [29], generalizing them to a context of first-class components. For-comprehensions are based on Haskell's monad comprehensions [44], even though their syntax more closely resembles XQuery [3]. Views have been influenced by Haskell's type classes [45]. They can be seen as an object-oriented version of parametric type classes [37], 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 [36]. 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 [39] 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 [12] 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[19] 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.

14 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's specification and implementation also indicate that its complexity is manageable. The current Scala compiler frontend is roughly as large as Sun's Java 1.4 frontend – we expect to decrease its size significantly by rewriting it completely in Scala. The current Scala specification [34] (about 100 pages) is considerably smaller than the current Java 1.4 specification [22] (about 400 pages). These sizes are hard to compare, though, as the Scala specification still lacks the level of maturity of the Java specification, and also uses shorter formulas in many places where the Java specification uses prose.

Scala has been released publicly on the JVM platform in January 2004 and on the .NET platform in June 2004. The implementation is complete except for run-time types; these are expected for the end of 2004. In the future, we intend to experiment with more refined type systematic support for XML, constructor polymorphism, and interfaces to database query languages, and to extend the current set of standard Scala libraries. We also plan to continue work on formalizing key aspects of of the language and on developing compiler optimizations targeted at its constructs.

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, Microsoft Research, and the Hasler Foundation. We also thank Gilad Bracha, Erik Ernst, Benjamin Pierce, Mads Torgersen, and Philip Wadler for useful discussions on aspects of the language.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang.* Prentice-Hall, second edition, 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, 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] 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.

[13] D. Duggan. Mixin modules. In *ACM SIGPLAN International Conference on Functional Programming*, 1996.

[14] ECMA. C# Language Specification. Technical Report Standard ECMA-334, 2nd Edition, December 2002.

[15] ECMA. Common Language Infrastructure. Technical Report Standard ECMA-335, 2nd Edition, 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] D. C. Fallside, editor. XML Schema. W3C recommendation, World Wide Web Consortium, May 2001. Available online `http://www.w3.org/TR/xmlschema-0/`.

[19] 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.

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

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

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

[23] 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.

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

[25] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications*, volume 34(10), pages 132–146, 1999.

[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 and al. The Scala Language Specification. Technical report, EPFL Lausanne, Switzerland, Jan. 2004. Available online `http://scala.epfl.ch`.

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

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

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

[43] 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.

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

[45] 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.

[46] M. Zenger. Erweiterbare Übersetzer. Master's thesis, University of Karlsruhe, August 1998.

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

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

[49] M. Zenger and M. Odersky. Extensible Algebraic Datatypes with Defaults. In *Proceedings of the International Conference on Functional Programming*, Firenze, Italy, September 2001.