Scala: Uniting Functional and Object-Oriented Programming

Margus Freudenthal

December 16, 2010

Abstract

Scala is a programming language that incorporates features from both functional and object-oriented programming languages. Scala is strongly statically typed. This paper describes the main interesting features of the Scala language.

First, features such as abstract member types, mixin composition and selftype annotation are targeted at creating independent components that can be composed to larger components. Second, pattern matching is popular feature in functional languages. In Scala implementation, pattern matching can also be performed without exposing implementation of the data type. Third, implicit parameters and methods can be used to conveniently pass around context variables and implementations of various interfaces.

1 Introduction

Scala is a general-purpose programming language that integrates features of functional and object-oriented programming. One important design constraint for Scala is its seamless integration with the Java language and the JVM platform. Scala code can be called from Java code and vice versa. On the one hand, this tight integration is needed to gain wider acceptance. On the other hand, this has quite a big impact on the language because all the language constructs in Scala must somehow map to JVM and Java. In

essence, Scala can be thought of as a superset of Java[1] that adds functional programming features (first-class closures, currying, pattern matching) and better type system (traits, covariant types).

The rest of this section presents very brief overview of the Scala language. For more details, refer to [OCD+06] or [OSV08].

1.1 Object-Oriented Programming in Scala

Unlike Java, Scala is fully object-oriented. This means that all the primitive types (e.g., integers) are objects and all the usual operators (e.g., + and *) are methods in these types.

Objects in scala are classes that have only one instance. In the Java world, they correspond to singleton classes (or, alternatively, classes that contain only static methods and attributes).

Traits in Scala are essentially interfaces that can also contain method definitions (in addition to method declarations). They represent compromise between multiple inheritance and single inheritance. In Scala, a class can inherit from one class but from several traits. When mapping Scala code to Java, traits are compiled to Java interfaces. Traits are useful in mixin compositions (see section 2.2).

Scala supports **covariance** and **contravariance**. For example, the class *GenList* below defines a covariant list type with methods *isEmpty*, *head* and *tail*. The covariance is indicated by annotation "+" before the type parameter *T*. This annotation ensures that if *X* is subclass of *Y* then *GenList[X]* is subclass of *GenList[Y]*. See figure 1 on page 21 for full code of the *GenList* class and its subclasses.

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

[1] All the Java language constructs do not have the exact counterpart in Scala. However, these can be emulated using Scala constructs. For example, static variables and methods can be emulated using Scala objects.

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 classied as covariant for the types of immutable fields and method results, and contravariant for method argument types and upper type parameter bounds.

For example, if we wish to add to the previous *GenList* example a binary operation *prepend*, then the parameter will be in contravariant position. Therefore, this will have to be constrained with lower bound (denoted by ">:" operator). In this way, *prepend* will return type that is a supertype of T.

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

In addition to regular data members (methods and fields), Scala classes and traits can contain **type members**. In the simplest case, type members are simply aliases to existing types (similar to C++ member typedefs). Type members can also be abstract – type members declared in the base class can be implemented in the child classes. This is discussed in more detail in section 2.1.

1.2 Functional Programming in Scala

Scala offers many features that are traditionally associated with functional programming languages: first-class functions, higher-order functions, currying, streams, and for comprehensions. These features are well-integrated with object-oriented language core. Object methods can be used as functions as shown in the following code. The following expression returns list containing characters "o", "o", and "b".

```
List(1, 2, 3).map(‘‘foobar’’.charAt)
```

Functions are internally represented as objects. For example, consider the following definition for incrementer function.

```
val inc = (x: Int) => x + 1
```

This definition is expaneded into code that instantiates subclass of the abstract *scala.Function1[S, T]* from the standard library.

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

In fact, several classes from Scala's standard library inherit from *FunctionN* class. Examples are *Array*, *Map* and *Set*.

2 Component Programming with Scala

This section describes in more detail language features that are directed at constructing reusable components. The components here are classes and the focus is on how to assemble classes that provide parts of the desired functionality. The goal is to develop the classes separately, with few dependencies between them, and only combine the classes at the moment where the combined functionality is really required. More thorough discussion of these features can be found in [OZ05].

2.1 Abstract Member Types

Abstract member types provide a way to abstract over concrete types of the components. They can hide information about internals of the component. Abstract member types have some similarities to SML signatures.

Let's start with an example. The following code snippet declares abstract class *AbsCell* that has abstract type member T and abstract value member *init*.

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

These type members are instantiated in a child class as shown in the following code snippet. The type of the value *cell* is $AbsCell\ \{type\ T = Int\}$. The type alias $cell.T = Int$ is available to code accessing the *cell* value and therefore, operations specific to type T are legal.

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

Following from the *AbsCell* example, it is also possible to process objects of type *AbsCell* without knowing the exact value of type T. For example, consider the following code.

```
def reset(c: AbsCell) {
    c.set(c.init)
}
```

This works because the expression *c.init* has type $c.T$ and the method *c.set* has type $c.T$ => *Unit*. Because concrete argument type matches formal parameter type, the method call is type-correct.

The expression c.T is an example of **path-dependent type**. In general, path-dependent types are expressions in the form of $X_1. \ldots .X_n.t$ where the prefix $X_1, \ldots, X_n$ is an immutable value (X_1 is an immutable value and X_n, $n > 1$ are immutable fields of the previous value), and t is type member of this value. For any value p, Scala defines path-dependent type named $p.\mathbf{type}$. This type is called a **singleton type** that represents just the object denoted by p. In fact, path-dependent type $p.t$ can be expressed using singleton type as $p.type\#t$. The # here is a type selection operator – $A\#B$ means "type field B of type A". It corresponds to Java's nested class reference operator *OuterClass.InnerClass*.

Singleton types are useful for specifying return types of methods. For example, let's consider class C that stores an integer and provides method *incr*. D is a subclass of C that adds a *decr* method to decrement the integer.

```
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}
}
```

Because the return type of *incr* is specified as *this.type*, the actual return type is not C, but the class of *this* (the subclass of C that is instantiated at the run time). Thus, if p is of type D then the return type for *p.incr* is also D. Thus we can easily chain calls to methods *incr* and *decr* like in the following example.

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

The *this.type* construct is mainly useful for creating APIs that use method chaining. This allows the methods in the base class return *this* with the correct type and therefore avoid the downcasting that is illustrated in the following Java code snippet.

```
class C {
    protected int x = 0
    C incr() {x = x + 1; return this;}
}
class D extends C {
    D decr() {x = x - 1; return this;}
}
...
((D) new D().incr()).decr()
```

Member types and type parameters can both be used to achieve similar means. For example, the first example can also be written using type parameters.

```
abstract class AbsCell[T] {
    val init: T
    ...
}
```

In principle, member types and type parameters are orthogonal features, much like abstract methods and method parameters. The cases when to use one or the other option are also similar. For one or two parameters, it is convenient to use type parameters. However, if the class depends on several types, using abstract member types results in cleaner code.

2.2 Modular Mixin Composition

Mixin composition is a way to compose components that are aware of each other and cooperate. For example, mixins allow attaching code to some predefined points, somewhat like in aspect-oriented programming.

We will first present example of mixins. Consider the following *AbsIterator* trait.

```
trait AbsIterator {
    type T
    def hasNext: Boolean
    def next: T
}
```

This defines abstract identifier that can iterate over elements of type T. Next, we define class *RichIterator* that extends the *AbsIterator* with method *foreach* that calls its argument function for every element in the collection.

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

Here is a concrete iterator class that iterates over characters of a given string.

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

When combining the functionality of *StringIterator* and *RichIterator* into a single class, we use Scala's mixin-class composition mechanism. This allows the programmer to reuse delta of a class definition – all the new definitions that are not inherited. Let's look at the following example.

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

[a]The *args(0)* parameter to *StringIterator* class is not related to the mixin composition. It is just a way to pass constructor parameters to parent classes (somewhat similar to C++).

In this example, the *Iter* class is constructed using mixin composition of the parents *StringIterator* and *RichIterator*. The first parent is called the superclass of *Iter*, and the second parent is called a mixin. Next, we will present exact rules for mixin composition.

The classes reachable through transitive closure of the direct inheritance relation from a class C are called the base classes of C. Because of mixins, the inheritance relationship on base classes forms a directed acyclic graph. A linearization of that graph is defined as follows.

Definition 2.1: Let C be a class with parents C_n `with` ... `with` C_1. The *class linearization of* C, $\mathcal{L}(C)$ is defined as follows:

$$\mathcal{L}(C) = \{C\} \overrightarrow{+} \mathcal{L}(C_1) \overrightarrow{+} \ldots \overrightarrow{+} \mathcal{L}(C_n)$$

Here $\overrightarrow{+}$ denotes concatenation where elements of the right operand replace identical elements of the left operand:

$$\{a, A\} \overrightarrow{+} B = \begin{cases} a, (A \overrightarrow{+} B) & if a \notin B \\ A \overrightarrow{+} B & if a \in B \end{cases}$$

For example, the linearization of class *Iter* is {*Iter, RichIterator, StringIterator, AbsIterator, AnyRef, Any*}. Note that for class C, $\mathcal{L}(C)$ always contains linearization of its superclass as suffix. However, the order in which the mixin classes appear in the linearization is dependent on the order in which they appear in class definition.

A class can inherit members from all the base classes.

Definition 2.2: A *concrete member* of a class C is any concrete definition M in some class $C_i \in \mathcal{L}(C)$, except if there is a preceding class $C_j \in \mathcal{L}(C)$ where $j < i$ which defines a concrete member M' matching M.
An abstract member of a class C is any abstract definition M in some class $C_i \in \mathcal{L}(C)$, except if C contains already a concrete member M' matching M, or if there is a preceding class $C_j \in \mathcal{L}(C)$ where $j < i$ which defines an abstract member M' matching M.

This definition shows what members of class C are considered abstract and what members concrete. It also determines the overriding relationship between matching members of a class C. First, a concrete definition always overrides an abstract definition. Second, for definitions M and M' which are both concrete or both abstract, M overrides M' if M appears in a class that precedes (in the linearization of C) the class in which M' is defined.

In Scala, the super calls have different meaning than in Java or C#. Consider, for example, the following two traits. The *SyncIterator* wraps all iterators into *synchronized* block. The *LoggedIterator* logs all calls to the *next* method.

```
abstract class SyncIterator extends AbsIterator {
    abstract override def hasNext: Boolean =
        synchronized(super.hasNext)
    abstract override def next: T =
        synchronized(super.next)
}
abstract class LoggedIterator extends AbsIterator {
    abstract override def next: T = {
        val x = super.next
        println(x)
        x
    }
}
```

The important point here is that the *super.next* and *super.hasNext* calls in *SyncIterator* and *LoggedIterator* classes cannot refer to their statically determined base class *AbsIterator*, because these methods are abstract. Instead, super calls are resolved based on class linearization.

Definition 2.3: Consider an expression *super.M* in a base class *C* of *D*. To be type correct, this expression must refer statically to some member *M* of a parent class of *C*. In the context of *D*, the same expression then refers to a member *M'* which matches *M*, and which appears in the first possible class that follows *C* in the linearization of *D*.
For example, the classes *Iter2* and *Iter3* combine *SyncIterator* and *LoggedIterator* in different order.

```
class Iter2 extends StringIterator(someString)
                with SyncIterator with LoggedIterator
class Iter3 extends StringIterator(someString)
                with LoggedIterator with SyncIterator
```

The class *Iter2* inherits *next* method from *LoggedIterator*, the *super.next* call in this method refers to the *next* method in *SyncIterator*, whose *super.next* in turn refers the *next* method in *StringIterator*. Therefore, method calls are logged before synchronization. In class *Iter3*, the order of mixins is different and logging happens inside synchronized sections.

Note that although methods in classes *LoggedIterator* and *SyncIterator* contain implementation, they are not really implementations of iterator contract. Therefore, these methods are marked *abstract* and *override*. The *abstract* modifier indicates that the method is in fact abstract and that the mixin composition must contain concrete definition of this method in some class that follows this class in linearization. The *override* modifier is necessary because the method actually overrides a concrete method in the mixin composition.

2.3 Selftype Annotations

Selftype annotations are the means of attaching a programmer-defined type to *this*. This is a good way to express required services of a component at the level where it connects with other components.

Let's consider a class for describing graphs.

```
abstract class Graph {
    type Node <: BaseNode
    class BaseNode {
        def connectWith(n: Node): Edge =
            new Edge(this, n)             // Error!
    }
    class Edge(from: Node, to: Node) {
        def source = from
        def target = to
    }
}
```

This code does not compile because the type of the self-reference in the *connectWith* method is *BaseNode* instead of *Node*. We need to express the fact that the identity of class *BaseNode* has to be expressible as type *Node*. This can be done by creating an accessor method for *this* as shown on Figure 2 on page 22. Scala offers direct support for this pattern by allowing the programmer to specify the type of *this* explicitly.

```
abstract class Graph {
    type Node <: BaseNode
    class BaseNode requires Node {
        def connectWith(n: Node): Edge =
            new Edge(this, n)             // OK
    }
    class Edge(from: Node, to: Node) {...}
}
```

The construct "*class BaseNode requires Node*" specifies that when the abstract class *BaseNode* is actually instantiated, the actual class must be compatible with type *Node*. Note that the selftype does not have to be related to the class being defined.

3 Pattern Matching

Pattern matching is a popular feature in many functional programming languages (ML family, Haskell, Erlang). Although object-oriented languages in general have shunned pattern matching (because it breaks encapsulation and

exposes object's internals), Scala has explicit support for pattern matching on objects (for more details, see [EOW07]).

Let's consider an example about symbolic manipulation of expressions. We assume a hierarchy of classes, rooted in a base class *Expr* and containing classes for specific forms of expressions, such as *Mul* for multiplication operations, *Var* for variables, and *Num* for numeric literals. Different forms of expressions have different members: *Mul* has two members *left* and *right* denoting its left and right operand, whereas *Num* has a member *value* denoting an integer. Our goal is to express simplification rules, such as:

> *x+1* is replaced with *x*.

Two examples of idiomatic object-oriented solutions to this problem are shown in the appendix. Figure 3 on page 23 shows solution that uses the visitor pattern. Figure 4 on page 24 shows solution that tests for type of node and then uses type cast to extract the fields. It is easy to see that these solutions get quickly very complex or verbose, especially if dealing with nested patterns. The following snippet shows Scala solution using case classes and pattern matching. Case classes are regular classes which export their constructor parameters and provide builtin support for pattern matching.

```
// Class hierarchy:
trait Expr
case class Num(value : int) extends Expr
case class Var(name : String) extends Expr
case class Mul(left : Expr, right : Expr) extends Expr
// Simplification rule:
e match {
    case Mul(x, Num(1)) => x
    case _ => e
}
```

This code looks quite natural and resembles ML-style pattern matching.

It can be argued that this style exposes implementation details, such as properties *left* and *right* in the *Mul* class. This can be solved by using **extractors**. Following code shows object *Twice* which enables pattern matching of even numbers.

```
object Twice {
    def apply(x: Int) = x * 2
    def unapply(z: Int) =
        if (z % 2 == 0) Some(z / 2)
        else None
}
```

This object defines an *apply* function, which provides a new way to write integers: *Twice(x)* is now an alias for *x * 2*. Scala uniformly treats objects with *apply* methods as functions, inserting the call to *apply* implicitly. Thus, *Twice(x)* is really a shorthand for *Twice.apply(x)*.

The *unapply* method in *Twice* reverses the construction in a pattern match. It tests its integer argument *z*. If *z* is even, it returns *Some(z / 2)*. If it is odd, it returns *None*. The *unapply* method is implicitly applied in a pattern match, as in the following example, which prints "*42 is two times 21*":

```
val x = Twice(21)
x match {
    case Twice(y) =>
        println(x+" is two times "+y)
    case =>
        println("x is odd")
}
```

In this example, method *unapply* is called extractor because it extracts parts of the given type. Extractors can have any number of arguments. A nullary pattern corresponds to an *unapply* method returning a boolean. A pattern with more than one element corresponds to an *unapply* method returning an optional tuple.

The example code in Figure 5 on page 25 shows how extractors can be used to hide representation of complex numbers. Complex numbers are stored as objects of type *Cart*. However, object *Polar* provides constructor and extractor for complex number that uses polar coordinates.

In fact, case classes can be considered as syntactic sugar for ordinary classes that have

- automatically generated companion object (object that has the same name as the class) with *apply* and *unapply* methods;

- automatically generated accessor methods for constructor parameters;
- sensible default implementations of *equals*, *hashCode* and *toString* methods.

Figure 6 on page 26 shows desugaring of the following case class:

```
case class Mul(left: Expr, right: Expr) extends Expr
```

4 Implicits

This section discusses Scala's implicit parameters and implicit methods [OSV08]. These are a powerful mechanism to automatically select values based on their type. The first subsection describes implicit parameters to methods and shows that they offer functionality that is comparable to Haskell's type classes [OMO10]. The second subsection implicit conversions (also called views) and their applications.

4.1 Implicit Parameters

To explain, what implicits are, let's assume that we want to define generic function for sorting lists. The type of the function will be:

```
def sort[T](xs: List[T])(ord: Ord[T]): List[T]
```

The function is parameterized with type of the objects being sorted (T). It takes two parameters[2]: *xs* containing the list being sorted and comparison function *ord*, and returns the sorted list. The Ord[T] can be defined as follows:

```
trait Ord[T] {
    def compare(a: T, b: T): Boolean
}
```

To use the sorting function, we must define comparator for a given type. For example:

[2]To be exact, it also takes two parameter sets, each containing one parameter. Multiple parameter sets allow partial evaluation of functions.

```
// Comparator for ints
object intOrd extends Ord[Int] {
    def compare(a: Int, b: Int): Boolean =
        a <= b
}

// Invoke the sorting function
sort(List(3, 2, 1))(intOrd)
```

The problem with this code is that it is necessary to pass around *Ord* instances. This affects methods in the whole call chain that will now have to become aware of ordering function. Scala solves this problem with **implicit parameters**. Implicit parameters are essentially a way to pass type-based parameters to functions. If parameter is defined as implicit and the compiler finds implicit value in the scope of the caller with type that matches the parameter type, then it uses this implicit value as the parameter value. With implicits, the previous example would look like this (the differences from the previous code are highlighted with red). Note that the call to *sort* function needs only one parameter – the *intOrd* is automatically passed as the *ord* parameter.

```
// Sorting function
def sort[T](xs: List[T])(implicit ord: Ord[T]): List[T]

// Comparator for ints
implicit object intOrd extends Ord[Int] {...}

// Invoke the sorting function
sort(List(3, 2, 1))
```

When looking for an implicit value of type T, the compiler will consider implicit value definitions (definitions introduced by *implicit val*, *implicit object*, or *implicit def*), as well as implicit arguments that have type T and that are in scope locally (accessible without prefix) where the implicit value is required. Additionally, it will consider implicit values of type T that are defined in the types that are part of the type T, as well as in the companion objects of the base classes of these parts. The set of parts of a type T is determined as follows:

- for a compound type T_1 *with* ... *with* T_n, the union of the parts of T_i, and T,

- for a parameterized type S *[*T_1*; ... ;*T_n*]*, the union of the parts of S and the parts of T_i,
- for a singleton type *p:type*, the parts of the type of p,
- for a type projection $S\#U$, the parts of S as well as $S\#U$ itself, in all other cases, just T itself.

This use of implicits is very similar to Haskell's type classes. The following snippet presents the sorting function in Haskell.

```
-- Define the type class
class Ord a where
    eq :: a -> a -> Bool
    compare :: a -> a -> Bool
-- Instantiate the Ord class for integers
instance Ord Int where
    eq a b = compare a b && compare b a
    compare x y = x <= y
-- The sorting function
sort :: Ord a => [a] -> [a]
-- Invoke the function
sort [3, 1, 2]
```

The main difference with Haskell type classes and implicits is that Haskell allows only one instance for each type[3]. In Scala, it is possible to have several instances for any given type. If the current scope contains several implicit instances for the type, the compiler gives error. In this case, the programmer must explicitly specify the parameter value. For example, the sorting function can also be called by passing an anonymous class as a value for the *ord* parameter.

```
sort(List(3, 2, 1))(new Ord[Int] { ... })
```

4.2 Views

A *view* is an implicit value of function type that converts a type A to B. The function has the type A => B. A view is applied in two circumstances.

[3]However, GHC has support for overlapping instances and tries to select the instance that is most specific to a given type parameter.

- When a type A is used in a context where another type B is expected and there is a view in scope that can convert A to B.
- When a non-existent member m of a type A is referenced, but there is an in-scope view that can convert A to a B that has the m member.

The Scala standard library uses views extensively for extending the common Java type. For example, the implicitly imported *Predef* object defines the following methods:

```
implicit def augmentString(x: String): StringOps =
    new StringOps(x)
implicit def unaugmentString(x: StringOps): String =
    x.repr
```

StringOps is a wrapper for Java *String* class that provides useful additional methods, for example, the ability to treat strings as lists of characters. When calling these additional methods on Java strings, the strings are converted to StringOps using the *augmentString* method. When calling e.g., Java methods, the *StringOps* is converted back to normal Java *String* via the *unaugmentString* method.

Views make it possible to extend existing libraries without having to modify them. In Scala community, extending an existing closed library is referred to as *Pimp my Library* pattern [Ode06]. A good example of application of this pattern is Scala's standard library which seamlessly extends Java standard classes.

5 Conclusion

5.1 Scala Language Features

This paper describes more interesting features of Scala programming language and type system. Abstract member types offer a way to express services required by a component. In this sense, they are very similar to abstract methods. Mixin composition allows building components from smaller components, each implementing part of the overall functionality. Mixin composition can also be used to inject various wrappers into inheritance hierarchi, thus achieving results similar to dependecy weaving or aspect-oriented

programming. Selftype annotations allow the programmer to attach user-defined type to *this*. This can also be used to express required services for a component.

Pattern matching has long been feature in functional programming languages. Scala implements it also in an object-oriented language. Case classes can be used for pattern matching much in the same way as algebraic datatypes in ML-style languages. Scala also supports *unapply* pattern that can be used to hide exact representation of object when using pattern matching.

Implicit parameters and are a powerful way to pass type-based parameters to methods. This can be used e.g., for transparently passing around various context variables. Implicit parameters can be used in the same way as type classes in Haskell. Additionally, implicit methods can be used to create views of the object. This provides the means to add methods to classes that cannot be modified by the programmer.

5.2 Scala Type System

Scala offers powerful language constructs and type system that can express quite complicated concepts succinctly. It also succeeds in integrating the functional and object-oriented styles – code written in object-oriented style can be called from e.g., higher-order functions and vice versa.

Although Scala is strongly typed language, its type system does not restrict one from writing short and efficient programs. Except when interfacing with Java-based APIs, there is almost no need for type casting. The type system features described in this paper allow one to create independent modules that can be combined in a flexible manner. Scala type system is based on formal model (the νObj calculus [OCRZ03]).

5.3 Issues With Scala

The powerful language constructs can also be used to write hard-to-read code. In particular, implicits can result in code that is dificult to understand and debug because lot can happen behind the scenes (in this sense, it is

similar to powerful features in other languages, such as reflection in Java, open classes in Ruby, macros in Scheme).

The situation is not helped by "there's more than one way to do it" philosophy: Scala offers several ways to achieve the same goal, often sacrifying consistency in order to achieve short code that "automagically" does the right thing. To give a very simple example, methods with one parameter can be used as infix operators: "*a b c*" is equivalent to "*a.b(c)*", except when *b* ends in *:*. In that case, "*a b c*" is equivalent to "*c.b(a)*". The exception is necessary in order to make the list construction ("cons") operator *::* be a method of list, not of the first element.

The Java compatibility is also a double-edged sword. Although it is necessary to gain wider acceptance for the language, the Java compatibility often makes type system more complex than it would be if the language were to be designed without this requirement.

5.4 Scala in Practice

Scala is not just research language. It is actively being used for developing commercial software. For example, several high-profile web sites (LinkedIn, Twitter, FourSquare) use Scala. Although Scala seems to generate fair amount of buzz, the actual usage is currently not very high. For example, in open source project directory Ohloh, Scala amounts to 0.15% of projects, 0.2% of commits and 0.05% of lines of code (the corresponding percentages for the current leader Java are 8%, 13% and 15%). However, Scala's usage is currently rising (1.5 times per year, according to Ohloh) and it is possible that in future Scala will be serious alternative to Java.

References

[EOW07] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *ECOOP 2007 – object-oriented programming*, volume 4609 of *LNCS*, pages 273–298. Springer, 2007.

[OCD+06] Martin Odersky, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean Mcdirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias

Zenger. An overview of the Scala programming language (second edition). Technical Report LAMP-REPORT-2006-001, EPFL, 2006.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP 2003 – object-oriented programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer Berlin / Heidelberg, 2003.

[Ode06] Martin Odersky. Pimp my Library. `http://www.artima.com/weblogs/viewpost.jsp?thread=179766`, 9 October 2006.

[OMO10] Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type Classes as Objects and Implicits. In *OOPSLA/SPLASH'10*, 2010.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc, 26 November 2008.

[OZ05] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *OOPSLA '05: proceedings of the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, pages 41–57, New York, NY, USA, 2005. ACM.

A Code Samples

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

Figure 1: Full implementation of *GenList* class

```
abstract class Graph {
    type Node <: BaseNode
    class BaseNode {
        def connectWith(n: Node): Edge =
            new Edge(self, n)
        def self: Node
    }
    class Edge(from: Node, to: Node) {
        def source = from
        def target = to
    }
}
...
// Extends the Graph and implements the
// accessor self.
class XGraph extends Graph {
    class Node(...) extends BaseNode {
        def self: Node = this
        ...
    }
}
```

Figure 2: Manually coding selftype annotation

```
// Class hierarchy:
trait Visitor[T] {
  def caseMul(t : Mul): T = otherwise(t)
  def caseNum(t : Num): T = otherwise(t)
  def caseVar(t : Var): T = otherwise(t)
  def otherwise(t : Expr): T = throw new MatchError(t)
}
trait Expr {
  def matchWith[T](v : Visitor[T]): T
}
class Num(val value : int) extends Expr {
  def matchWith[T](v : Visitor[T]): T = v.caseNum(this)
}
class Var(val name : String) extends Expr {
  def matchWith[T](v : Visitor[T]): T = v.caseVar(this)
}
class Mul(val left : Expr, val right : Expr) extends Expr {
  def matchWith[T](v : Visitor[T]): T = v.caseMul(this)
}
// Simplification rule:
e.matchWith {
  new Visitor[Expr] {
    override def caseMul(m: Mul) =
      m.right.matchWith {
        new Visitor[Expr] {
          override def caseNum(n : Num) =
            if (n.value == 1) m.left else e
          override def otherwise(e : Expr) = e
        }
      }
    override def otherwise(e : Expr) = e
  }
```

Figure 3: Object-oriented pattern matching using visitors

```
// Class hierarchy:
trait Expr
class Num(val value : int) extends Expr
class Var(val name : String) extends Expr
class Mul(val left : Expr, val right : Expr) extends Expr
// Simplification rule:
if (e.isInstanceOf[Mul]) {
    val m = e.asInstanceOf[Mul]
    val r = m.right
    if (r.isInstanceOf[Num]) {
        val n = r.asInstanceOf[Num]
        if (n.value == 1) m.left else e
    } else e
} else e
```

Figure 4: Object-oriented pattern matching using type-test/type-cast

```
trait Complex
case class Cart(re: Double, im: Double) extends Complex
object Polar {
    def apply(mod: Double, arg: Double): Complex =
        new Cart(mod  Math.cos(arg), mod * Math.sin(arg))
    def unapply(z: Complex): Option[(Double, Double)] =
        z match {
            case Cart(re, im) =>
                val at = atan(im / re)
                Some(sqrt(re * re + im * im),
                    if (re < 0) at + Pi
                    else if (im < 0) at + Pi  2
                         else at)
    }
}
...
def printPolar(c: Complex) = c match {
    case Polar(mod, arg) =>
        println(``arg=''+arg+'', mod=''+mod)
    case _ =>
        ()
}
printPolar(Cart(1, 2))
printPolar(Polar(1, Pi))
```

Figure 5: Pattern matching on complex numbers

```
class Mul( left: Expr, right: Expr) extends Expr {
    // Accessors for constructor arguments
    def left = left
    def right = right

    // Standard methods
    override def equals(other: Any) = other match {
        case m: Mul =>
            left.equals(m.left) && right.equals(m.right)
        case _ => false
    }
    override def hashCode =
        hash(this.getClass, left.hashCode, right.hashCode)
    override def toString =
        "Mul("+left+", "+right+")"
}
object Mul {
    def apply(left: Expr, right: Expr) =
        new Mul(left, right)
    def unapply(m: Mul) =
        Some(m.left, m.right)
}
```

Figure 6: Expansion of case class *Mul*