The Scala Experiment – Can We Provide Better Language Support for Component Systems?

(Invited Talk Abstract)

Martin Odersky
EPFL

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language constructs and features – Classes and objects; modules; polymorphism; recursion.
General Terms: Languages

1 Introduction

Progress in component systems has been slowed down by shortcomings in the programming languages used to define and integrate components. In particular, the following tasks are often difficult to perform with today's mainstream technologies:

- Extending a system with new data variants as well as new operations,
- combining several extensions of a common base,
- abstracting over required services of a component,
- defining libraries that are as usable as domain-specific languages.

With Scala [3] we aimed to develop a language that makes type-safe component abstraction and composition simple and natural. Our work started from two hypotheses: First, languages for components need to be *scalable*, in that the same concepts should be able to describe small as well as large parts. Second, this form of scalability can be achieved by unifying and generalizing constructs from functional and object-oriented programming.

Scala is both an object-oriented and functional language. It is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes. Classes can be composed using mixin composition. Scala is designed to work seamlessly with two less pure but mainstream object-oriented languages – Java and C#.

Scala is a functional language in the sense that every function is a value. Nesting of function definitions and higher-order functions are naturally supported. Scala also supports a general notion of pattern matching which can model the algebraic types used in many functional languages.

In the following, we outline with three small examples how a tight integration of functional and object-oriented programming leads to synergies and enables new constructions that were so far difficult to express.

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.
1-59593-027-2/06/0001

```
abstract class Term[T];
case class Lit(x: int) extends Term[int];
case class Succ(t: Term[int]) extends Term[int];
case class IsZero(t: Term[int]) extends Term[boolean];
case class If[T](c: Term[boolean],
                 t1: Term[T],
                 t2: Term[T]) extends Term[T];

def eval[T](t: Term[T]): T = t match {
 case Lit(n)        => n
  case Succ(u)       => eval(u) + 1
  case IsZero(u)     => eval(u) == 0
  case If(c, u1, u2) => eval(if (eval(c)) u1 else u2)
}
```

Listing 1: A statically typed evaluator

2 Objects and Pattern Matching

Many functional languages have algebraic data types, with values that are decomposed via pattern matching. Object-oriented languages have instead class hierarchies and virtual methods. It is well-known that the two notions have complementary strengths and weaknesses when it comes to extending a system: The functional approach makes it easy to add new operations, whereas the object-oriented approach makes it easy to add new data variants. Scala combines both approaches using case classes. A case class is like a normal class except that instance constructors can be recovered by pattern matching. Figure 1 shows as an example a typed evaluator of simple arithmetic expressions with conditionals.

There is an abstract base class `Term` that represents arithmetic expressions. Every term carries in its type parameter `T` the type of the expression it denotes. The subclasses `Lit` and `Succ` denote terms of type `int`, whereas the subclass `isZero` denotes a term of type `boolean`. The last subclass, `If`, is itself polymorphic; it can denote a term of type `int` or `boolean` depending on its arguments.

The second half of Figure 1 shows an evaluator of expressions. Function `eval` takes a term `t` of type `Term[T]` and returns a value of type `T`. It operates by a pattern match on the possible constructors of a term.

In this example, the object-oriented inheritance gives us for free a heterogeneous type hierarchy where type parameters can vary in subclasses. Functional pattern matching provides a simple way to decompose terms and to recover the heterogeneity of the type arguments. The combination of both techniques gives the full power of GADT's [6] which have been recently explored in functional languages.

```
class Auction(seller: Actor, minBid: int, closing: Date)
extends Actor {
  override def run() = {
    var maxBid = 0;
    var maxBidder: Actor = null;
    var running = true;
    while (running) {
      val now = new Date();
      val remaining = closing.getTime() - now.getTime();
      receiveWithin (remaining) {
        case Offer(bid, client) =>
          if (bid > maxBid) {
            maxBid = bid; maxBidder = client;
          }
        case Inquire(client) =>
          client send Status(maxBid, maxBidder, closing)
        case TIMEOUT =>
          running = false;
          seller send Status(maxBid, maxBidder, closing)
      }
    }
  }
}
```

Listing 2: An auction class

3 Objects and Functions

Scala is an object-oriented language in that every value is an object. It is also a functional language in that functions are first-class values. It follows that functions in Scala must be themselves objects. For instance, a unary function from `S` to `T` is an instance of the standard Scala class `scala.Function1[S, T]`, which is defined as follows:

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

As can be seen from this definition, functions are objects with `apply` methods. The apply method invocation can be elided in a function call; i.e. if `f` is a function object then `f(e)` is equivalent to `f.apply(e)`. The signs in front of the type parameters of class `Function1` are *variance annotations*. They specify that functions are contravariant in their argument type and covariant in their result type.

Because function types are classes, it is possible to form specialized subclasses. One obvious specialization are arrays, which are mutable functions over the integers with additional `length` and `update` methods. Another specialization are *partial functions* which have besides the apply method also a method `isDefinedAt` which tests whether a function is defined for a given argument. Pattern matching blocks such as the one in the body of the `eval` function of Figure 1 are instances of partial functions: they can be applied to a selector value, and one can also test whether there is a pattern which matches a given selector value.

An interesting application of partial functions is message-based process communication in the style of Erlang [1]. The Scala libraries defines a class of `Actor`s, which are threads that communicate via messages. Messages are instances of arbitrary case classes. The `a.send(m)` method call sends a message `m` to an actor `a`. Messages are queued in a mailbox, which the receiving actor can query using methods `receive` and `receiveWithin`. Both of these methods take as argument a partial function that maps patterns of messages to actions. They will select from the mailbox the first queued message that matches any of the patterns in the partial function.

As an example of this style of process communication, Listing 2 presents an auction process that communicates with bidders via messages `Offer` and `Inquire`. The process repeatedly invokes a `receiveWithin` method that waits for messages and processes them. A special `TIMEOUT` message is generated once the closing date of the auction is reached.

This style of process communication has been extensively applied in Erlang projects. The point we make with Scala is that a tight integration of object-oriented and functional constructs enables new ways of expression that make Erlang's native operations definable in a library. For this it is necessary to combine first-class functions with function type specialization via inheritance.

4 Objects and Modules

Traditionally, object systems and module systems are separate entities. Module systems are used to define and link components whereas object systems are used to define component implementations. The problem with this approach is its scalability. The result of a complicated component composition might be seen as a simple object or class on the next level of program integration. For instance, the Scala compiler itself results from the composition of some fairly complicated components, but seen as an Eclipse [2] plugin, it is a class which can be instantiated – possibly several times – to a compiler object.

To address scenarios like this, Scala unifies the object and module systems. Components are classes that can be composed using mixin composition. Required components and services are expressed as abstract members of a class, or by abstracting over the type of self in a class. Objects (i.e. class instances) take on the double duty of modules.

This style of component abstraction and composition is enabled by three programming language constructs: abstract type members, explicit selftypes, and modular mixin composition. All three constructs have their foundation in the νObj calculus [4]. Together, they enable us to transform an arbitrary assembly of static program parts with hard references between them into a system of reusable components. The transformation maintains the structure of the original system. A detailed account of these constructions is given in a separate paper [5].

5 REFERENCES

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[2] Object Technology International. *Eclipse Platform Technical Overview*, Feb. 2003. `www.eclipse.org`.

[3] M. Odersky *et al.* An Overview of the Scala Programming Language, TR IC/2004/64, EPFL.

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

[5] M. Odersky and M. Zenger. Scalable component abstractions. In *Proc. OOPSLA 2005*, Oct. 2005.

[6] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proc. 30th Symp. on Principles of Programming Languages*, pages 224–235, January 2003.