Scala Rationale

Martin Odersky

March 17, 2006

There are hundreds of programming languages in active use, and many more are being designed each year. It is therefore hard to justify the development of yet another language. Nevertheless, this is what we attempt to do here. Our argument is based on two claims:

> *Claim 1:* The raise in importance of web services and other distributed software is a fundamental paradigm shift in programming. It is comparable in scale to the shift 20 years ago from character-oriented to graphical user interfaces.
>
> *Claim 2:* That paradigm shift will provide demand for new programming languages, just as graphical user interfaces promoted the adoption of object-oriented languages.

For the last 20 years, the most common programming model was object-oriented: System components are objects, and computation is done by method calls. Methods themselves take object references as parameters. Remote method calls let one extend this programming model to distributed systems. The problem of this model is that it does not scale up very well to wide-scale networks where messages can be delayed and components may fail. Web services address the message delay problem by increasing granularity, using method calls with larger, structured arguments, such as XML trees. They address the failure problem by using transparent replication and avoiding server state. Conceptually, they are *tree transformers* that consume incoming message documents and produce outgoing ones.

Why should this have an effect on programming languages? There are at least two reasons: First, today's object-oriented languages are not very good at analyzing and transforming XML trees. Because such trees usually contain data but no methods, they have to be decomposed and constructed from the "outside", that is from code which is external to the tree definition itself. In an object-oriented language, the ways of doing so are limited. The most common solution [W3C] is to represent trees in a generic way, where all tree nodes are values of a common type. This makes it easy to write

generic traversal functions, but forces applications to operate on a very low conceptual level, which often loses important semantic distinctions present in the XML data. More semantic precision is obtained if different internal types model different kinds of nodes. But then tree decompositions require the use of run-time type tests and type casts to adapt the treatment to the kind of node encountered. Such type tests and type casts are generally not considered good object-oriented style. They are rarely efficient, nor easy to use.

By contrast, tree transformation is the natural domain of functional languages. Their algebraic data types, pattern matching and higher-order functions make these languages ideal for the task. It's no wonder, then, that specialized languages for transforming XML data such as XSLT are functional.

Another reason why functional language constructs are attractive for web-services is that mutable state is problematic in this setting. Components with mutable state are harder to replicate or to restore after a failure. Data with mutable state is harder to cache than immutable data. Functional language constructs make it relatively easy to construct components without mutable state.

Many web services are constructed by combining different languages. For instance, a service might use XSLT to handle document transformation, XQuery for database access, and Java for the "business logic". The downside of this approach is that the necessary amount of cross-language glue can make applications cumbersome to write, verify, and maintain. A particular problem is that cross-language interfaces are usually not statically typed. Hence, the benefits of a static type system are missing where they are needed most – at the join points of components written in different paradigms.

Conceivably, the glue problem could be addressed by a "multi-paradigm" language that would express object-oriented, concurrent, as well as functional aspects of an application. But one needs to be careful not to simply replace cross-language glue by awkward interfaces between different paradigms within the language itself. Ideally, one would hope for a fusion which unifies concepts found in different paradigms instead of an agglutination, which merely includes them side by side. This fusion is what we try to achieve with Scala [1].

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 mainstream object-oriented languages, in particular Java and C#.

[1] Scala stands for "Scalable Language". The term means "Stairway" in Italian

Scala is also 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. Furthermore, this notion of pattern matching naturally extends to the processing of XML data.

The design of Scala is driven by the desire to unify object-oriented and functional elements. Here are three examples how this is achieved:

- Since every function is a value and every value is an object, it follows that every function in Scala is an object. Indeed, there is a root class for functions which is specialized in the Scala standard library to data structures such as arrays and hash tables.

- Data structures in many functional languages are defined using algebraic data types. They are decomposed using pattern matching. Object-oriented languages, on the other hand, describe data with class hierarchies. Algebraic data types are usually closed, in that the range of alternatives of a type is fixed when the type is defined. By contrast, class hierarchies can be extended by adding new leaf classes. Scala adopts the object-oriented class hierarchy scheme for data definitions, but allows pattern matching against values coming from a whole class hierarchy, not just values of a single type. This can express both closed and extensible data types, and also provides a convenient way to exploit run-time type information in cases where static typing is too restrictive.

- Module systems of functional languages such as SML or Caml excel in abstraction; they allow very precise control over visibility of names and types, including the ability to partially abstract over types. By contrast, object-oriented languages excel in composition; they offer several composition mechanisms lacking in module systems, including inheritance and unlimited recursion between objects and classes. Scala unifies the notions of object and module, of module signature and interface, as well as of functor and class. This combines the abstraction facilities of functional module systems with the composition constructs of object-oriented languages. The unification is made possible by means of a new type system based on path-dependent types [OCRZ03].

There are several other languages that try to bridge the gap between the functional and object oriented paradigms. Smalltalk[GR83], Python[vRD03], or Ruby[Mat01] come to mind. Unlike these languages, Scala has an advanced static type system, which contains several innovative

constructs. This aspect makes the Scala definition a bit more complicated than those of the languages above. On the other hand, Scala enjoys the robustness, safety and scalability benefits of strong static typing. Furthermore, Scala incorporates recent advances in type inference, so that excessive type annotations in user programs can usually be avoided.

References

[GR83] Adele Goldberg and David Robson. *Smalltalk-80; The Language and Its Implementation*. Addison-Wesley, 1983. ISBN 0-201-11371-6.

[Mat01] Yukihiro Matsumoto. *Ruby in a Nutshell*. O'Reilly & Associates, nov 2001. ISBN 0-596-00214-9.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. FOOL 10*, January 2003. `http://www.cis.upenn.edu/~bcpierce/FOOL/FOOL10.html`.

[vRD03] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd, sep 2003. ISBN 0-954-16178-5 `http://www.python.org/doc/current/ref/ref.html`.

[W3C] W3C. Document object model (DOM). `http://www.w3.org/DOM/`.