The Scala Experiment

–

Can We Provide Better Language Support for Component Systems?

Martin Odersky
EPFL

Component software – state of the art

In *principle*, software should be constructed from re-usable parts (“components”).

In *practice*, software is still most often written “from scratch”, more like a craft than an industry.

Programming languages share part of the blame.

Most existing languages offer only limited support for components.

This holds in particular for statically typed languages such as Java and C#.

How to do better?

Hypothesis 1: Languages for components need to be *scalable*; the same concepts should describe small as well as large parts.

Hypothesis 2: Scalability can be achieved by unifying and generalizing *functional* and *object-oriented* programming concepts.

Why unify FP and OOP?

Both have complementary strengths for composition:

Functional programming: Makes it easy to build interesting things from simple parts, using

- higher-order functions,
- algebraic types and pattern matching,
- parametric polymorphism.

Object-oriented programming: Makes it easy to adapt and extend complex systems, using

- subtyping and inheritance,
- dynamic configurations,
- classes as partial abstractions.

These are nice reasons, but...

In reality, I no longer wanted to sit between two chairs!

Scala

Scala is an object-oriented and functional language which is completely interoperable with Java.

(the .NET version is currently under reconstruction.)

It removes some of the more arcane constructs of these environments and adds instead:

(1) a uniform object model,

(2) pattern matching and higher-order functions,

(3) novel ways to *abstract* and *compose* programs.

An open-source distribution of Scala has been available since Jan 2004.

Currently: $\approx$ 1000 downloads per month.

Unifying FP and OOP

In the following, I present three examples where

- ▷ formerly separate concepts in FP and OOP are identified, and
- ▷ the fusion leads to something new and interesting.

Scala unifies

- algebraic data types with class hierarchies,
- functions with objects,
- modules with objects.

1^{st} Unification: ADTs are class hierarchies

Many functional languages have algebraic data types and pattern matching.

$\Rightarrow$ Concise and canonical manipulation of data structures.

Object-oriented programmers object:

- “ADTs are not extensible!”
- “ADTs violate the purity of the OO data model!”
- “Pattern matching breaks encapsulation!”

Pattern matching in Scala

- Here's a a set of definitions describing binary trees:

```
abstract Tree[T]
case object Empty extends Tree
case class Binary(elem : T, left : Tree[T], right : Tree[T]) extends Tree
```

- And here's an inorder traversal of binary trees:

```
def inOrder[T](t : Tree[T]): List[T] = t match {
    case Empty          ⇒ List()
    case Binary(e, l, r)  ⇒ inOrder(l) ::: List(e) ::: inOrder(r)
}
```

- The case modifier of a class means you can pattern match on it.

Pattern matching in Scala

- Here's a a set of definitions describing binary trees:

```
abstract Tree[T]
case object Empty extends Tree
case class Binary(elem : T, left : Tree[T], right : Tree[T]) extends Tree
```

- And here's an inorder traversal of binary trees:

```
def inOrder[T](t : Tree[T]): List[T] = t match {
  case Empty          ⇒ List()
  case Binary(e, l, r) ⇒ inOrder(l) ::: List(e) ::: inOrder(r)
}
```

- This design keeps:

Purity:	All cases are classes or objects.
Extensibility:	You can define more cases elsewhere.
Encapsulation:	Only constructor parameters of case classes are revealed.

2^{nd} Unification: functions are objects

Scala is a functional language, in the sense that every function is a value.

Functions can be anonymous, curried, nested.

Familiar higher-order functions are implemented as methods of Scala classes. E.g.:

```
matrix.exists(row ⇒ row.forall(0 ==)))
```

Here, matrix is assumed to be of type Array[Array[int]], using Scala's Array class (explained below)

If functions are values, and values are objects, it follows that functions themselves are objects.

In fact, the function type $S \Rightarrow T$ is equivalent to

```
scala.Function1[S, T]
```

where Function1 is defined as follows in the standard Scala library:

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

(Analogous conventions exist for functions with more than one argument.)

Hence, functions are interpreted as objects with apply methods. For example, the anonymous "successor" function x: int ⇒ x + 1 is expanded as follows.

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

Why should I care?

Since ⇒ is a class, it can be subclassed.

So one can *specialize* the concept of a function.

An obvious use is for arrays – mutable functions over integer ranges.

```
class Array[A](length : int) extends (int ⇒ A)  {
    def length : int = ...
    def apply(i : int): A = ...
    def update(i : int, x : A): unit = ...
    def elements : Iterator[A] = ...
    def exists(p : A ⇒ boolean): boolean = ...
}
```

Another bit of syntactic sugaring lets one write:

```
a(i) = a(i) ∗ 2     for     a.update(i, a.apply(i) ∗ 2)
```

Partial functions

Another useful abstraction are *partial functions*.

These are functions that are defined only in some part of their domain.

What's more, one can inquire with the isDefinedAt method whether a partial function is defined for a given value.

```
abstract class PartialFunction[-A, +B] extends (A => B) {
  def isDefinedAt(x: A): Boolean
}
```

Scala treats blocks of pattern matching cases as instances of partial functions.

This lets one express control structures that are not easily expressible otherwise.

Example: Erlang-style actors

Two principal constructs (adopted from Erlang).

```
actor ! message                       // asynchronous message send

receive {                             // message receieve
    case msgpat$_1$ ⇒ action$_1$
    ...
    case msgpat$_n$ ⇒ action$_n$
}
```

Send is asynchronous; messages are buffered in an actor's mailbox.

receive picks the first message in the mailbox which matches any of the patterns mspat$_i$.

If no pattern matches, the actor suspends.

Example: orders and cancellations

```
val orderManager =
    actor {
        loop {
            receive {
                case Order(sender, item) ⇒
                    val o = handleOrder(sender, item); sender ! Ack(o)
                case Cancel(o : Order) ⇒
                    if (o.pending) { cancelOrder(o); sender ! Ack(o) }
                    else sender ! NoAck
                case x ⇒
                    junk += x
        }}}
val customer = actor {
    orderManager ! myOrder
    ordermanager receive { case Ack ⇒ ... }
}
```

Implementing receive

Using partial functions, it is straightforward to implement receive:

```
        def receive[A](f: PartialFunction[Message, A]): A = {
          self.mailBox.extractFirst(f.isDefinedAt) match {
            case Some(msg) ⇒ f(msg)
            case None ⇒ self.wait(messageSent)
          }
        }
        ...
      }
```

Here,

  self designates the currently executing actor,
  mailBox is its queue of pending messages, and
  extractFirst extracts first queue element matching given predicate.

Library or language?

A possible objection to Scala's library-based approach is:

> *Why define actors in a library when they exist already in purer, more optimized form in Erlang?*

One good reason is that libraries are much easier to extend and adapt than languages.

For instance, both Erlang and the Scala library attach one thread to each Actor.

This is a problem for Java, where threads are expensive.

Erlang is *much* better at handling many threads, but even it can be overwhelmed by huge numbers of actors.

Event-based actors

An alternative are event-based actors.

Normally, this means inversion of control, with a global rewrite of the program.

But if actors are implemented as a library, it is easy to implement a variation of receive (call it react) which liberates the running thread when it blocks for a message.

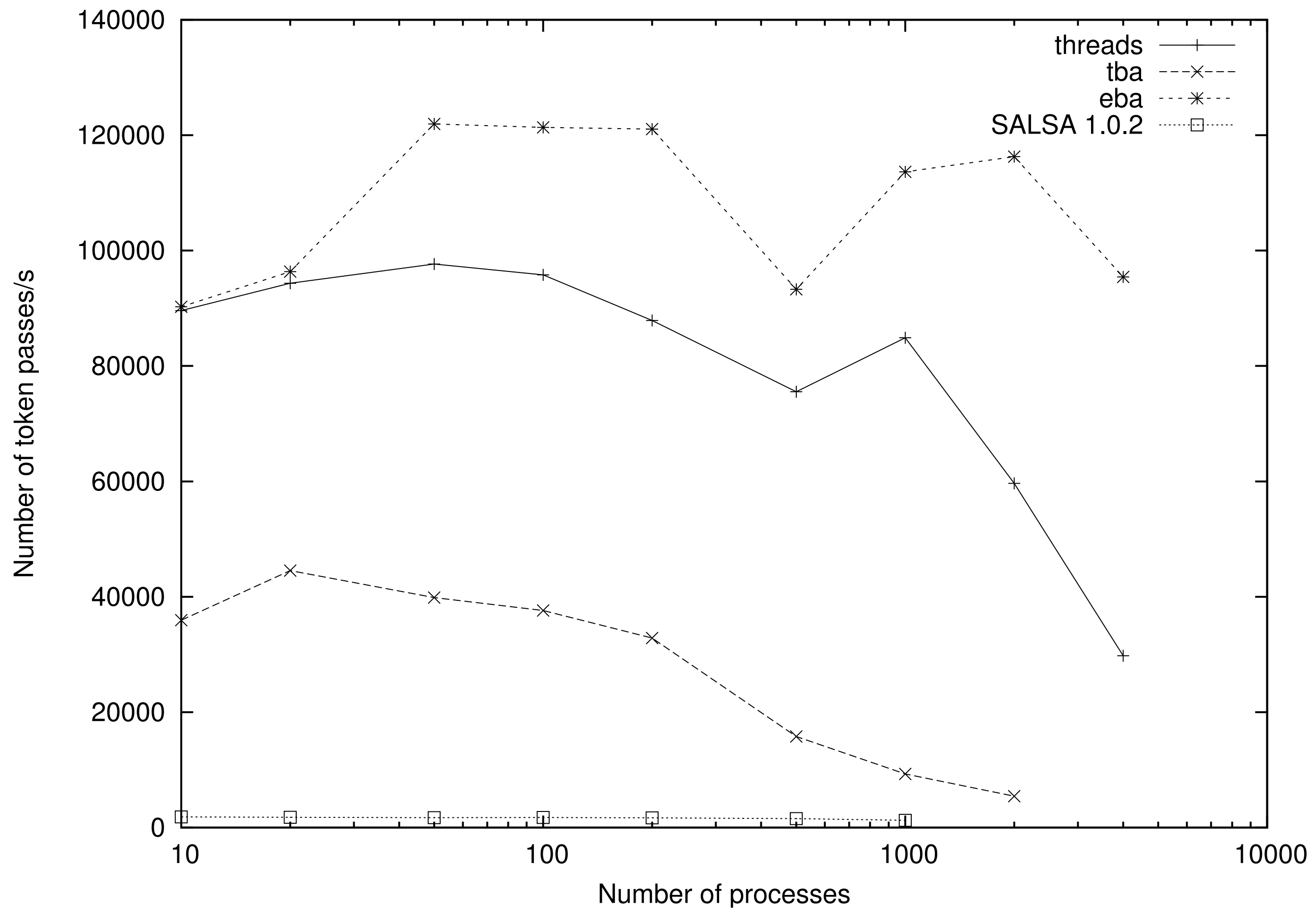
The only restriction is that react should never return normally:

```
def react(f: PartialFunction[Message, unit]): Nothing = ...
```

Client-code is virtually unchanged between the multi-threaded and event-based versions of the library.

Performance: react vs. receive

Number of token passes per second in a ring of processes.



3^{rd} Unification: modules are objects

Scala has a clean and powerful type system which enables new ways of abstracting and composing components.

A *component* is a program part, to be combined with other parts in larger applications.

Requirement: Components should be *reusable*.

To be reusable in new contexts, a component needs *interfaces* describing its *provided* as well as its *required* services.

Most current components are not very reusable.

Most current languages can specify only provided services, not required services.

Note: Component $\neq$ API !

No statics!

A component should refer to other components not by hard links, but only through its required interfaces.

Another way of expressing this is:

> *All references of a component to others should be via its members or parameters.*

In particular, there should be no global static data or methods that are directly accessed by other components.

This principle is not new.

But it is surprisingly difficult to achieve, in particular when we extend it to classes.

Functors

One established language abstraction for components are SML functors.

Here,

Component	$\hat{=}$	*Functor or Structure*
Interface	$\hat{=}$	*Signature*
Required Component	$\hat{=}$	*Functor Parameter*
Composition	$\hat{=}$	*Functor Application*

Sub-components are identified via sharing constraints.

Shortcomings:

- No recursive references between components
- No inheritance with overriding
- Structures are not first class.

Components in Scala

In Scala:

Component	$\hat{=}$	*Class*
Interface	$\hat{=}$	*Abstract Class*
Required Component	$\hat{=}$	*Abstract Type Member or*
		Explicit Self-Type
Composition	$\hat{=}$	*Modular Mixin Composition*

Advantages:

+ Components instantiate to objects, which are first-class values.
+ Recursive references between components are supported.
+ Inheritance with overriding is supported.
+ Sub-components are identified by name
  $\Rightarrow$ no explicit "wiring" is needed.

Component abstraction

There are two principal forms of abstraction in programming languages:

> parameterization (functional)
>
> abstract members (object-oriented)

Scala supports both styles of abstraction for types as well as values.

Both types and values can be parameters, and both can be abstract members.

(In fact, Scala works with the *functional/OO duality* in that parameterization can be expressed by abstract members).

Abstract types

Here is a type of “cells” 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) { value = x }
}
```

The AbsCell class 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.

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

The type of cell is AbsCell { type T = int }.

Path-dependent types

- It is also possible to access AbsCell without knowing the binding of its type member.
- For instance: `def reset(c: AbsCell): unit = c.set(c.init);`
- Why does this work?
  - ▷ c.init has type c.T
  - ▷ The method c.set has type c.T $\Rightarrow$ unit.
  - ▷ So the formal parameter type and the argument type coincide.

c.T is an instance of a *path-dependent* type.

In general, such a type has the form $x_0. \ldots .x_n.t$, where

- x_0 is an immutable value
- $x_1, \ldots, x_n$ are immutable fields, and
- t is a type member of x_n.

Safety requirement

Path-dependent types rely on the immutability of the prefix path.

Here is an example where 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!
```

Scala's type system does not admit the last 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.

Example: symbol tables

Here's an example which reflects a learning curve I had when writing extensible compiler components.

- Compilers need to model symbols and types.
- Each aspect depends on the other.
- Both aspects require substantial pieces of code.

The first attempt of writing a Scala compiler in Scala defined two global objects, one for each aspect:

First attempt: global data

```
object Symbols {
    class Symbol {
        def tpe : Types.Type;
        ...
    }
    // static data for symbols
}
```

```
object Types {
    class Type {
        def sym : Symbols.Symbol
        ...
    }
    // static data for types
}
```

Problems:

- Symbols and Types contain hard references to each other.

  Hence, impossible to adapt one while keeping the other.
- Symbols and Types contain static data.

  Hence the compiler is not *reentrant*, multiple copies of it cannot run in the same OS process.

  (This is a problem for the Scala Eclipse plug-in, for instance).

Second attempt: nesting

Static data can be avoided by nesting the Symbols and Types objects in a common enclosing class:

```
class SymbolTable {
    object Symbols {
        class Symbol { def tpe : Types.Type; ... }
    }
    object Types {
        class Type {def sym : Symbols.Symbol; ... }
    }
}
```

This solves the re-entrancy problem. But it does not solve the component reuse problem.

- Symbols and Types still contain hard references to each other.
- Worse, they can no longer be written and compiled separately.

Third attempt: a component-based solution

Question: How can one express the required services of a component?

Answer: By abstracting over them!

Two forms of abstraction: *parameterization* and *abstract members*.

Only abstract members can express recursive dependencies, so we will use them.

```
abstract class Symbols {
  type Type
  class Symbol { def tpe : Type }
}
```

```
abstract class Types {
  type Symbol
  class Type { def sym : Symbol }
}
```

Symbols and Types are now classes that each abstract over the identity of the "other type". How can they be combined?

Modular mixin composition

Here's how:

```
class SymbolTable extends Symbols with Types
```

Instances of the SymbolTable class contain all members of Symbols as well as all members of Types.

Concrete definitions in either base class override abstract definitions in the other.

[Modular mixin composition generalizes the single inheritance + interfaces concept of Java and C#.
It is similar to *traits* [Schaerli et al, ECOOP 2003], but is more flexible since base classes may contain state.]

Fourth attempt: mixins + self-types

The last solution modeled required types by abstract types.

In practice this can become cumbersume, because we have to supply (possibly large) interfaces for the required operations on these types.

A more concise approach makes use of *self-types*:

```
class Symbols requires Symbols with Types {
  class Symbol { def tpe : Type }
}
class Types requires Types with Symbols {
  class Type { def sym : Symbol }
}
class SymbolTable extends Symbols with Types
```

Here, every component has a *self-type* that contains all required components.

Self-types

- In a class declaration

  ```
  class C requires T { ... }
  ```

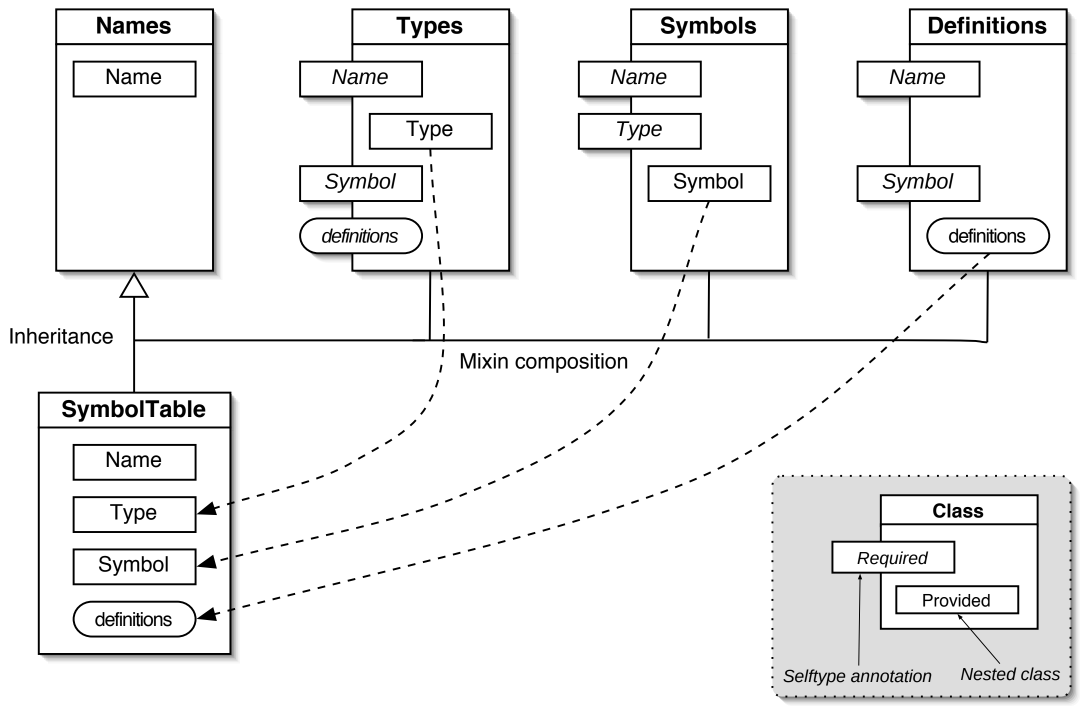
  T is called a *self-type* of class C.
- If a self-type is given, it is taken as the type of this inside the class.
- Without an explicit type annotation, the self-type is taken to be the type of the class itself.

Safety Requirement

- The self-type of a class must be a subtype of the self-types of all its base classes.
- 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.

Symbol table schema

Here's a schematic drawing of *scalac*'s symbol table:



We see that besides Symbols and Types there are several other classes that also depend recursively on each other.

Benefits

- The presented scheme is very *general* – any combination of static modules can be lifted to a assembly of components.
- Components have *documented interfaces* for required as well as provided services.
- Components can be *multiply instantiated*

  ⇒ *Re-entrancy* is no problem.
- Components can be flexibly *extended* and *adapted*.

Example: logging

As an example of component adaptation, consider adding some logging facility to the compiler.

Say, we want a log of every symbol and type creation, written to standard output.

The problem is how insert calls to the methods for logging into an existing compiler

- without changing source code,
- with clean separation of concerns,
- without using AOP.

Logging classes

The idea is that the tester of the compiler would create subclasses of components which contain the logging code. E.g.

```
abstract class LogSymbols extends Symbols {
    override def newTermSymbol(name: Name): TermSymbol = {
        val x = super.newTermSymbol(name)
        System.out.println("creating term symbol " + name)
        x
    }
    ...
}
```

... and similarly for LogTypes.

How can these classes be integrated in the compiler?

Inserting behavior by mixin composition

Here's an outline of the Scala compiler root class:

```
class ScalaCompiler extends SymbolTable with ... { ... }
```

To create a logging compiler, we extend this class as follows:

```
class TestCompiler extends ScalaCompiler with LogSymbols with LogTypes
```

Now, every call to a factory method like newTermSymbol is re-interpreted as a call to the corresponding method in LogSymbols.

Note that the mixin-override is non-local – methods are overridden even if they are defined by indirectly inherited classes.

Mixins + self-types *vs.* AOP

Similar strategies work for many adaptations for which aspect-oriented programming is usually proposed. E.g.

- security checks
- synchronization
- choices of data representation (e.g. sparse vs dense arrays)

Generally, one can handle all before/after advice on method join-points in this way.

However, there's no quantification over join points as in AOP.

Summing up

Here are counts of non-comment lines of three compilers I was involved with:

old scalac[1]	48,484 loc
new scalac[2]	22,823 loc
javac[3]	28,173 loc

Notes:

[1] Written in Pizza (30824 loc) and Scala (17660 loc), Scala code was literally translated from Pizza.
[2] Written in Scala.
[3] Pre-wildcards version of javac 1.5, written in Java.

This indicates that Scala achieves a compression in source code size of more than 2 relative to Java.

Things that worked out well:

- Combination of FP and OOP was much richer than anticipated:
  - GADTs,
  - type classes, concepts,
  - Scala compiler was a fertile testing ground for new design patterns.
- Scala lets one write pleasingly concise and expressive code.
- It shows great promise as a host language for DSL's.
- New discoveries on the OOP side:
  - modular mixin composition,
  - selftypes,
  - type abstraction and refinement.

Immersion in the Java/.NET world was a double-edged sword:

+ It helped adoption of the language.
+ It saved us the effort of writing a huge number of libraries.
+ We could ride the curve in performance improvements of the VM implementations (quite good for 1.5 JVM's by IBM and Sun).

– It restricted the design space of the language:
  - No true virtual classes.
  - Had to adopt overloading, null-references as is.
  - Limits on tail recursion.

– It made the language design more complex.

– It prevented some bitsy optimizations.

It would be interesting to see another unification of FP and OOP that is less determined by existing technology.

Conclusion

- Scala blends functional and object-oriented programming.
- This has worked well in the past: for instance in Smalltalk, Python, or Ruby.
- However, Scala is the first to unify FP and OOP in a statically typed language.
- This leads to pleasant and concise programs.
- Programming in Scala has a similar feel as programming in a modern "scripting language", but without giving up static typing.

Try it out: scala.epfl.ch

Thanks to the (past and present) members of the Scala team:

Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Philipp Haller, Sean McDermid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, Matthias Zenger.

Relationship between Scala and other languages

Main influences on the Scala design:

- Java, C# for their syntax, basic types, and class libraries,
- Smalltalk for its uniform object model,
- Beta for systematic nesting,
- ML, Haskell for many of the functional aspects.
- OCaml, OHaskell, PLT-Scheme, as other combinations of FP and OOP.
- Pizza, Multi-Java, Nice as other extensions of Java with functional ideas.

(Too many influences in details to list them all)

Scala also seems to influence other new language designs, see for instance the closures and comprehensions in C# 3.0.

Related language research

Mixin composition: Bracha (linear), Duggan, Hirschkowitz (mixin-modules), Schaerli et al. (traits), Flatt et al. (units, Jiazzi), Zenger (Keris).

Abstract type members: Even more powerful are *virtual classes* (Ernst, Ostermann, Cook)

Explicit self-types: Vuillon and Rémy (OCaml)

Scala and XML

A rich application field for functional/object-oriented languages is XML input and output.

Reason:

- XML data are trees which need to be accessed “from the outside”.

  Functional languages are particularly good at this task.

- At the same time, XML data are often processed by systems that are dynamically composed (e.g. web services).

  Object-oriented languages are particularly good at this task.

However, to be useful, a language needs some minimal support for XML.

Scala provides:

- XML literals with holes)
- XML regular expression pattern matching
- However, it does not provide regular expression types.