The Scala Experiment

—

Can We Provide Better Language Support for Component Systems?

Martin Odersky, EPFL

Invited Talk,
ACM Symposium on Programming Languages and Systems (POPL),
Charleston, South Carolina, January 11, 2006.

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.

An Experiment

To validate our hypotheses we have designed and implemented a concrete programming language, **Scala**.

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

Currently: $\approx$ 500 downloads per month.

Version 2 of the language has been released 10 Jan 2006.

A language by itself proves little; its applicability can only be validated by practical, serious use.

Scala

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

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.

Interoperability

Scala is completely interoperable with Java (and with some qualifications also to C#).

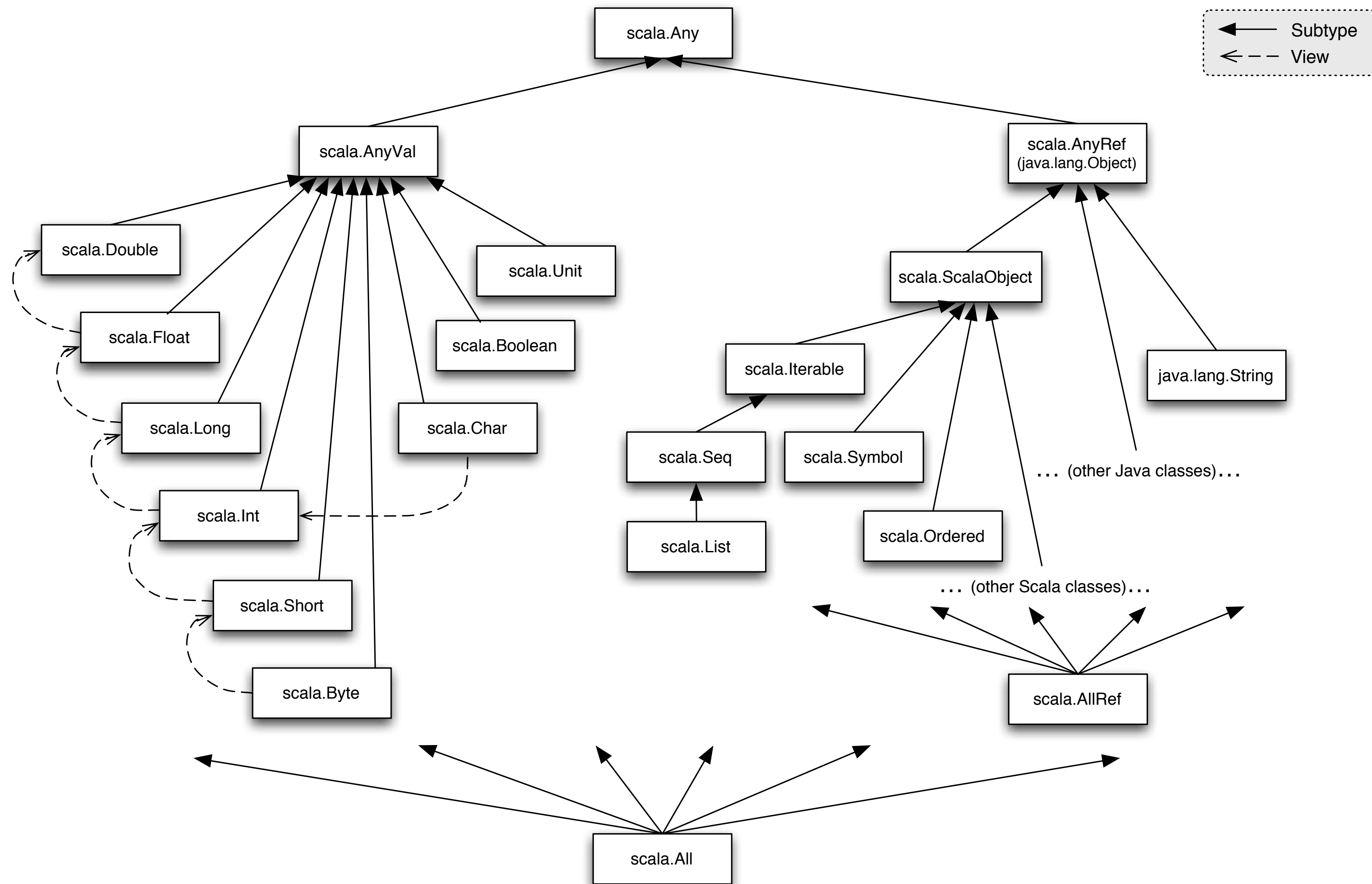
A Scala component can:

- access all methods and fields of a Java component,
- create instances of Java classes,
- inherit from Java classes and implement Java interfaces,
- be itself instantiated and called from a Java component.

None of this requires glue code or special tools.

This makes it very easy to mix Scala and Java components in one application.

Scala's Class Hierarchy



A Unified Object Model

In Scala, every value is an object and every operation is a method invocation.

Example: A class for natural numbers

```
abstract class 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
}
```

Here are the two canonical implementations of Nat:

```
class Succ(n : Nat) extends Nat {
  def isZero : boolean = false;
  def pred : Nat = n
}
```

```
object Zero extends Nat {
  def isZero : boolean = true;
  def pred : Nat = error("Zero.pred")
}
```

Higher-Order Functions

- 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 could be of type of List[List[int]], using Scala's List class:

```
class List[+T] {
  def isEmpty : boolean;
  def head : T
  def tail : List[T]
  def exists(p : T ⇒ boolean): boolean =
    !isEmpty && (p(head) || (tail exists p))
  ...
}
```

Unifying FP and OOP

In the following, I show three concrete examples where

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

Scala unifies

1. algebraic data types with class hierarchies,
2. functions with objects,
3. modules with objects.

1st 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

Scala sees algebraic data types as special cases of class hierarchies.

- No separate algebraic data types – every type is a class.
- Can pattern match directly over classes.
- A pattern can access the constructor parameters of a (case) class.

Example (1): A Class Hierarchy of Terms

```
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]
```

- The **case** modifier in front of a class means you can pattern match on it.
- Note that some subclasses *instantiate* the type parameter T.
- One cannot describe this hierarchy using a plain old ADT, but a GADT would do.

Example (2): A Typed Evaluator

```
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]


// Note that eval instantiates T differently for each case */

def eval[T](t: Term[T]): T = t match {
  case Lit(n)          ⇒ n                                // T = int
  case Succ(u)         ⇒ eval(u) + 1                      // T = int
  case IsZero(u)       ⇒ eval(u) == 0                     // T = boolean
  case If(c, t1, t2)   ⇒ eval(if (eval(c)) t1 else t2)
}
```

2nd Unification: Functions are Objects

- 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 "incrementer" 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 $\Rightarrow$ 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] = ...
}
```

Another bit of syntactic sugar powdering 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

Erlang defines processes that communicate via *mailboxes*.

One can send a message to an actor's mailbox.

A receive primitive serves to retrieve messages that match any of a given set of patterns.

Example:

```
receive {
  case Order(sender, item) ⇒
    val o = handleOrder(sender, item); sender.send(Ack(o))
  case Cancel(o: Order) ⇒
    cancelOrder(o)
  case x ⇒
    junk += x
}
```

Implementing receive

Using partial functions, it is straightforward to implement receive as a method of the Actor class:

```
class Actor {
  protected def receive[A](f: PartialFunction[Message, A]): A = {
    pendingMsgs.find(f.isDefinedAt) match {
      case Some(msg) => dequeue(msg); f(msg)
      case None => wait(messageSent)
    }
  }
  ...
}
```

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.

But if actors are implemented as a library, it is easy to write another mailbox library, where receive's are executed at the expense of the sender's thread.

The only restriction is that receive should never return normally:

```
def receive(f: PartialFunction[Message, T]): scala.All = ...
```

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

Part 2: 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,

$$\begin{array}{lcl} \textit{Component} & \hat{=} & \textit{Functor} \text{ or } \textit{Structure} \\ \textit{Interface} & \hat{=} & \textit{Signature} \\ \textit{Required Component} & \hat{=} & \textit{Functor Parameter} \\ \textit{Composition} & \hat{=} & \textit{Functor Application} \end{array}$$

Sub-components are identified via sharing constraints.

Shortcomings:

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

3rd Unification: Modules Are Objects

In Scala:

Component	$\hat{=}$	*Class*
Interface	$\hat{=}$	*Abstract Class*
Required Component	$\hat{=}$	*Abstract Member* or *"Self"*
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.

Language Constructs for Components

Scala has three concepts which are particularly interesting in component systems.

- *Abstract type members* allow to abstract over types that are members of objects.
- *Self-type annotations* allow to abstract over the type of "self".
- *Modular mixin composition* provides a flexible way to compose components and component types.

All three abstractions have their theoretical foundation in the νObj calculus [Odersky et al., ECOOP03].

They subsume SML modules.

More precisely, (generative) SML modules can be encoded in νObj, but not *vice versa*.

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): unit = { 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 ⇒ 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:

1. Symbols and Types contain hard references to each other.

   Hence, impossible to adapt one while keeping the other.

2. 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 a 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, since they are nested in an enclosing object 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.

This is limiting, because one cannot instantiate or inherit an abstract type.

A more general approach also 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 { ... }

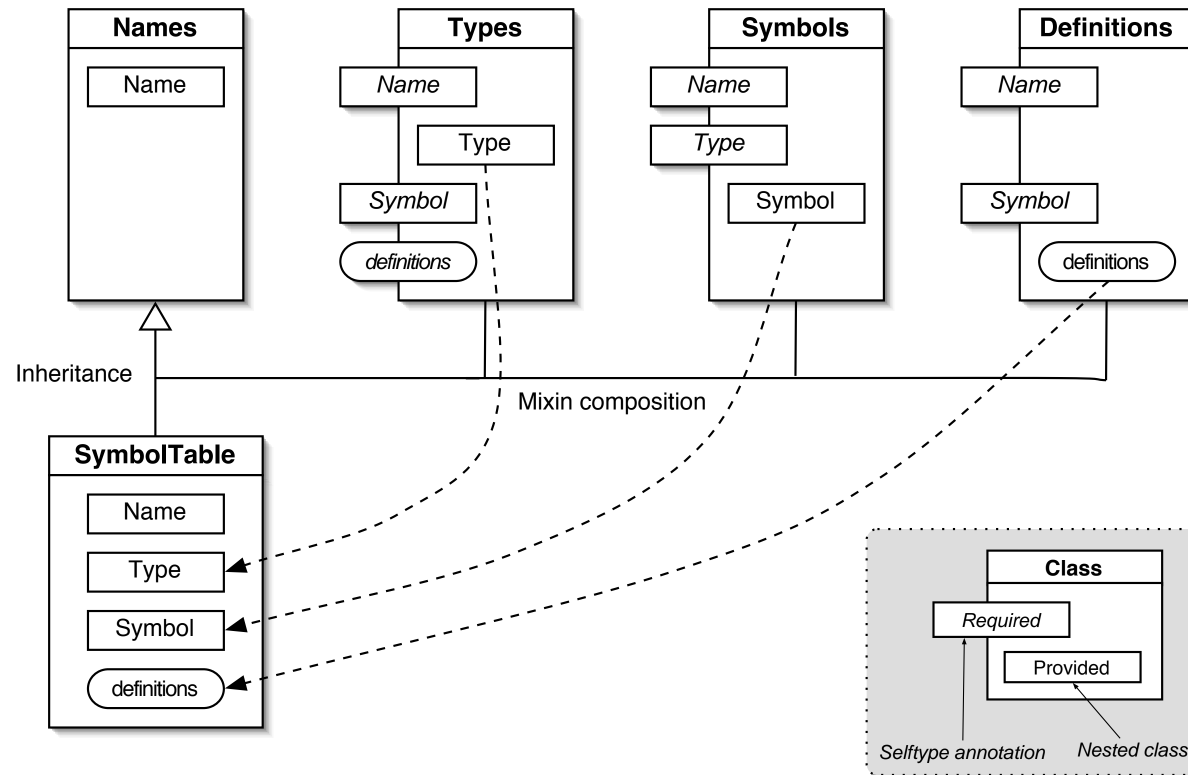
  T is called a *self-type* of class C.
- If a self-type is given, it is taken as the type of **this** inside the class.
- Without an explicit type annotation, the self-type is taken to be the type of the class itself.

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

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

   $\Rightarrow$ *Re-entrancy* is no problem.
4. 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

Some numbers:

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.
- New discoveries on the OOP side:
  - modular mixin composition,
  - selftypes,
  - type abstraction and refinement.

Things that worked out less well than expected:

- Type rules produced some gotcha's, have been tightened up in version 2.

  Example:

  ```
  val s: String = ""
  s.length == 0          // is always false!
  ```

- Regular expression patterns: hard to implement but not yet used much.
- Powerful import clauses make name space control more difficult.

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.

Things that remain to be done:

- Formalizations
  - There is νObj [ECOOP03].
  - But we would like to have a calculus which is closer to the Scala language, with decidable type checking.
  - We have drafts of a “Featherweight Scala”, but meta-theory remains to be fleshed out.
- Domain specific extensions
  - Scala shows great promise as a host language for DSL’s.
  - Joint work with Jagadeesan, Nadathur, Saraswat on embedding recursive constraint programming.
- Optimizing compilers
- Libraries, applications
- Standards
- Others?

Contributions On Any of These Are Very Welcome!

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* (Cook, Ernst, Ostermann in this conference)

Explicit self-types : Vuillon and Rémy (OCaml)

Conclusion

- Despite 10+ years of research, there are still interesting things to be discovered at the intersection of functional and object-oriented programming.
- Much previous research concentrated on simulating *some of* X in Y, where $X, Y \in \{\text{FP, OOP}\}$.
- More things remain to be discovered if we look at symmetric combinations.
- Scala is one attempt to do so.

  Try it out: scala.epfl.ch

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

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