Scalable Component Abstractions

Martin Odersky
EPFL
CH-1015 Lausanne
martin.odersky@epfl.ch

Matthias Zenger
Google Switzerland GmbH
Freigutstrasse 12
CH-8002 Zürich
zenger@google.com

ABSTRACT

We identify three programming language abstractions for the construction of reusable components: abstract type members, explicit selftypes, and modular mixin composition. Together, these abstractions 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. We demonstrate this approach in two case studies, a subject/observer framework and a compiler front-end.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language constructs and features – Classes and objects; inheritance; modules; packages; polymorphism; recursion.

General Terms

Languages

Keywords

Components, classes, abstract types, mixins, Scala.

1. INTRODUCTION

True component systems have been an elusive goal of the software industry. Ideally, software should be assembled from libraries of pre-written components, just as hardware is assembled from pre-fabricated chips or pre-defined integrated circuits. In reality, large parts of software applications are often written "from scratch," so that software production is still more a craft than an industry.

Components in this sense are simply program parts which are used in some way by larger parts or whole applications. Components can take many forms; they can be modules, classes, libraries, frameworks, processes, or web services. Their size might range from a couple of lines to hundreds of thousands of lines. They might be linked with other components by a variety of mechanisms, such as aggregation, parameterization, inheritance, remote invocation, or message passing.

An important requirement for components is that they are *reusable*; that is, that they should be applicable in contexts other than the one in which they have been developed. Generally, one requires that component reuse should be possible without modifiying a component's source code. Such modifications are undesirable because they have a tendency to create versioning problems. For instance, a version conflict might arise between an adaptation of a component in some client application and a newer version of the original component. Often, one goes even further in requiring that components are distributed and deployed only in binary form [43].

To enable safe reuse, a component needs to have *interfaces* for provided as well as for required services through which interactions with other components occur. To enable flexible reuse in new contexts, a component should also minimize "hard links" to specific other components which it requires for its functioning.

We argue that, at least to some extent, the lack of progress in component software is due to shortcomings in the programming languages used to define and integrate components. Most existing languages offer only limited support for component abstraction and composition. This holds in particular for statically typed languages such as Java [16] and C# [9] in which much of today's component software is written. While these languages offer some support for attaching interfaces describing the provided services of a component, they lack the capability to abstract over the services that are required. Consequently, most software modules are written with hard references to required modules. It is then not possible to reuse a module in a new context that refines or refactors some of those required modules.

Ideally, it should be possible to lift an arbitrary system of software components with static data and hard references, resulting in a system with the same structure, but with neither static data nor hard references. The result of such a lifting should create components that are first-class values. We have identified three programming language abstractions that enable such liftings.

Abstract type members provide a flexible way to abstract over concrete types of components. Abstract types can hide information about internals of a component, similar to their use in SML signatures. In an object-oriented framework where classes can be extended by inheritance, they may also be used as a flexible means of parameterization (often called *family polymorphism* [11]).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
Copyright 2005 ACM 1-59593-031-0/05/0010 ...$5.00.

Selftype annotations allow one to attach a programmer-defined type to `this`. This turns out to be a convenient way to express required services of a component at the level where it connects with other components.

Modular mixin composition provides a flexible way to compose components and component types. Unlike functor applications, mixin compositions can establish recursive references between cooperating components. No explicit wiring between provided and required services is needed. Services are modelled as component members. Provided and required services are matched by name and therefore do not have to be associated explicitly by hand.

All three abstractions have their theoretical foundation in the νObj calculus [35]. They have been defined and implemented in the programming language Scala. We have used them extensively in a component-oriented rewrite of the Scala compiler, with encouraging results.

The three abstractions are *scalable*, in the sense that they can describe very small as well as very large components. Scalability is ensured by the principle that the result of a composition should have the same fundamental properties as its constituents. In our case, components correspond to classes, and the result of a component composition is always a class again, which might have abstract members and a selftype annotation, and which might be composed with other classes using mixin composition. Classes on every level can create objects (also called "runtime components") which are first-class values, and therefore are freely configurable.

Related work

The concept of functor [27, 17, 24] in the module systems of SML [17] and OCaml [24], provides a way to abstract over required services in a statically type-checked setting. It represents an important step towards true component software. However, functors still pose severe restrictions when it comes to structuring components. Recursive references between separately compiled components are not allowed and inheritance with dynamic binding is not available.

ML modules, as well as other component formalisms [1, 30, 42, 51] introduce separate layers that distinguish between components and their constituents. This approach might have some advantages in that each formalism can be tailored to its specific needs, and that programmers receive good syntactic guidance. But it limits scalability of component systems. After all, what is a complicated system on one level might be a simple element on the next level of scale. For instance, the Scala compiler itself is certainly a non-trivial system, but it is treated simply as an object when used as a plugin for the Eclipse [33] programming environment. Furthermore, different instantiations of the compiler might exist simultaneously at runtime. For example, one instantiation might do a project rebuild, while another one might do a syntax check of a currently edited source file. Those instantiations of the compiler should have no shared state, except for the Eclipse runtime environment and the global file system. In a system where the results of a composition are not objects or classes, this is very hard to achieve.

Scala's aim to provide advanced constructs for the abstraction and composition of components is shared by several other research efforts. From Beta [28] comes the idea that everything should be nestable, including classes. To address the problem of expressing nested structures that span several source files, Beta provides a "fragment system" as a mechanism for weaving programs, which is outside the language proper. This is similar to what is done in aspect-oriented programming (indeed, the fragment system has been used to emulate AOP [23]).

Abstract types in Scala have close resemblances to abstract types of signatures in the module systems of SML and OCaml, generalizing them to a context of first-class components. Abstract types are also very similar to the virtual types [29] of the Beta and gbeta languages. In fact, virtual types in Beta can be modelled precisely in Scala by a combination of abstract types and selftype annotations. Virtual types as found in gbeta are more powerful than either Scala's or Beta's constructions, since they can be inherited as superclasses. This opens up possibilities for advanced forms of class hierarchy reuse [12], but it makes it very hard to check for accidental and incompatible overrides. Closely related are also the delegation layers of Caesar [38, 31], FamilyJ's virtual classes [48] and the work on nested inheritance for Java [32].

Scala's design of mixins comes from object-oriented linear mixins [3], but defines mixin composition in a symmetric way, similar to what is found in mixin modules [8, 18] or traits [41]. Jiazzi [30] is an extension of Java that adds a module mechanism based on *units* [15], a powerful form of parametrized modules. Jiazzi supports extensibility idioms similar to Scala, such as the ability to implement mixins. Jiazzi is built on top of Java, but its module language is not integrated with Java and therefore is used more like a separate language for linking Java code.

OCaml [25] and Moby [13] are both languages that combine functional and object-oriented programming using static typing. Unlike Scala, these two languages start with a rich functional language including a sophisticated module system and then build on these a comparatively lightweight mechanism for classes.

The only close analogue to selftype annotations in Scala is found in OCaml, where the type of *self* is an extensible record type which is explicitly given or inferred. This gives OCaml considerable flexibility in modelling examples that are otherwise hard to express in statically typed languages. But the context in which selftypes are used is different in both languages. Instead of subtyping, OCaml uses a system of parametric polymorhism with extensible records. The object system and module systems in OCaml are kept separate. Since selftypes are found only in the object system, they play a lesser role in component abstraction than in Scala.

The rest of this paper is structured as follows. Section 2 introduces Scala's programming constructs for component abstraction and composition. Section 3 shows how these constructs are applied in a type-safe subject/observer framework. Section 4 discusses a larger case study where the Scala compiler itself was transformed into a system with reusable components. Section 5 discusses lessons learned from the case studies. Section 6 concludes.

2. CONSTRUCTS FOR COMPONENT ABSTRACTION AND COMPOSITION

This section introduces the language constructs of Scala insofar as they are necessary to understand the cases stud-

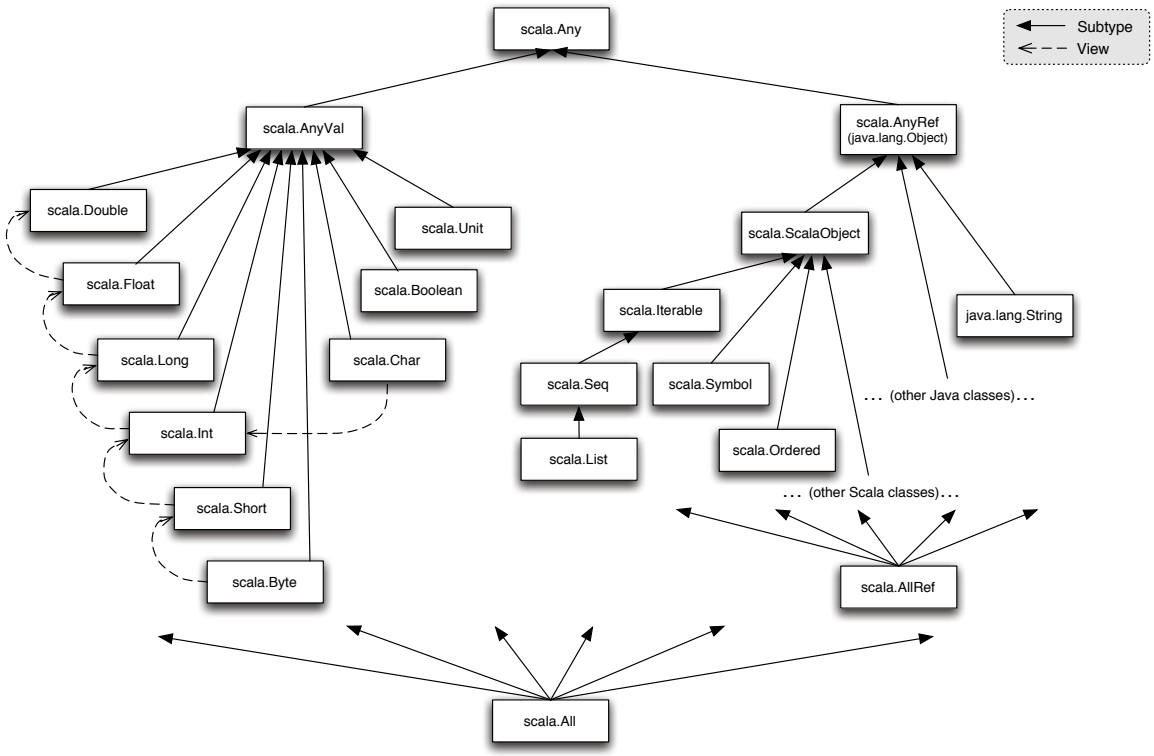

Figure 1: Standard Scala classes.

ies that follow. Scala fuses object-oriented and functional programming in a statically typed language. Conceptually, it builds on a Java-like core, even though its syntax differs. To this foundation, several extensions are added.

From the object-oriented tradition comes a uniform object model, where every value is an object and every operation is a method invocation. From the functional tradition come the ideas that functions are first-class values, and that some objects can be decomposed using pattern matching. Both traditions are merged in the conception of a novel type system, where classes can be nested, classes can be aggregated using mixin composition, and where types are class members which can be either concrete or abstract.

Scala provides full interoperability with Java. Its programs are compiled to JVM bytecodes, with the .NET CLR as an alternative implementation. Figure 1 shows how primitive types and classes of the host environment are integrated in Scala's class graph. At the top of this graph is class `Any`, which has two subclasses: Class `AnyVal`, from which all value types are derived and class `AnyRef`, from which all reference types are derived. The latter is identified with the root class of the host environment (`java.lang.Object` for the JVM or `System.Object` for the CLR). At the bottom of the graph are class `All`, which has no instances, and class `AllRef`, which has the **`null`** reference as its only instance. Note that value classes do not have `AllRef` as a subclass and consequently do not have **`null`** as an instance. This makes it possible to map value classes in Scala to the primitive types of the host environment.

Space does not permit us to present Scala in full in this paper; for this, the reader is referred elsewhere [34]. In this section we focus on a description of Scala's language constructs that are targeted to component design and composition. We concentrate our presentation on Scala 2.0, which differs in some details from previous versions. The description given here is informal. A theory that formalizes Scala's key constructs and proves their soundness is provided by the νObj calculus [35].

2.1 Abstract Type Members

An important issue in component systems is how to abstract from required services. There are two principal forms of abstraction in programming languages: parameterization and abstract members. The first form is typical for functional languages, whereas the second form is typically used in object-oriented languages. Traditionally, Java supports parameterization for values, and member abstraction for operations. The more recent Java 5.0 with generics supports parameterization also for types.

Scala supports both styles of abstraction uniformly for types as well as values. Both types and values can be parameters, and both can be abstract members. The rest of this section gives an introduction to object-oriented abstraction in Scala and reviews at the same time a large part of its type system. We defer a discussion of functional type abstraction (*aka* generics) to the appendix, because this aspect of the language is more conventional and not as fundamental for composition in the large.

To start with an example, the following class defines cells of values that can be read and written.

```
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 defines neither type nor value parameters. Instead it 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 in subclasses. The following program shows how to do this in Scala using an anonymous class.

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

The type of value `cell` is `AbsCell { type T = int }`. Here, the class type `AbsCell` is augmented by the *refinement* `{ type T = int }`. This makes the type alias `cell.T = int` known to code accessing the `cell` value. Therefore, operations specific to type `T` are legal, e.g. `cell.set(cell.get * 2)`.

Path-dependent types

It is also possible to access objects of type `AbsCell` without knowing the concrete binding of its type member. For instance, the following method resets a given cell to its initial value, independently of its value type.

```
def reset(c: AbsCell): unit = c.set(c.init);
```

Why does this work? In the example above, the expression `c.init` has type `c.T`, and the method `c.set` has function type `c.T => unit`. Since the formal parameter type and the concrete argument type coincide, the method call is type-correct.

`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 $n \geq 0$, x_0 denotes an immutable value, each subsequent x_i denotes an immutable field of the path prefix $x_0.\ \ldots\ .x_{i-1}$, and t denotes a type member of the path $x_0.\ \ldots\ .x_n$.

Path-dependent types rely on the immutability of the prefix path. Here is an example where this 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!
```

In this example subsequent calls to `f()` return cells where the value type is alternatingly either `int` or `String`. The last statement in the code above is erroneous since it tries to set an `int` cell to a `String` value. The type system does not admit this statement, because the computed type of `f().get` would be `f().T`. This type is not well-formed, since the method call `f()` does not constitute a well-formed path.

Type selection and singleton types

In Java, where classes can also be nested, the type of a nested class is denoted by prefixing it with the name of the outer class. In Scala, this type is also expressible, in the form of `Outer#Inner`, where `Outer` is the name of the outer class in which class `Inner` is defined. The "`#`" operator denotes a *type selection*. Note that this is conceptually different from a path dependent type p`.Inner`, where the path p denotes a value, not a type. Consequently, the type expression `Outer#`t is not well-formed if t is an abstract type defined in `Outer`.

In fact, path dependent types can be expanded to type selections. The path dependent type $p.t$ is taken as a shorthand for p`.type#`t. Here, p`.type` is a *singleton type*, which represents just the object denoted by p. Singleton types by themselves are also useful in other contexts, for instance they facilitate chaining of method calls. As an example, consider a class `C` with a method `incr` which increments a protected integer field, and a subclass `D` of `C` which adds a `decr` method to decrement that field.

```
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 }
}
```

Then we can chain calls to the `incr` and `decr` method, as in

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

Without the singleton type **`this.type`**, this would not have been possible, since `d.incr` would be of type `C`, which does not have a `decr` member. In that sense, **`this.type`** is similar to (covariant uses of) Kim Bruce's *mytype* construct [5].

Parameter bounds

We now refine the `Cell` class so that it also provides a method `setMax` which sets a cell to the maximum of the cell's current value and a given parameter value. We would like to define `setMax` so that it works for all cell value types admitting a comparison operation "`<`", which is a method of class `Ordered`. For the moment we assume this class is defined as follows (a more refined generic version of this class is in the standard Scala library).

```
abstract class Ordered {
  type O;
  def < (that: O): boolean;
  def <= (that: O): boolean =
    this < that || this == that
}
```

Class `Ordered` has a type "`O`" and a method "`<`" as abstract members. A second method, "`<=`", is defined in terms of "`<`". Note that Scala does not distinguish between operator names and normal identifiers. Hence, "`<`" and "`<=`" are legal method names. Furthermore, infix operators are treated as method calls. For identifiers m and operand expressions e_1, e_2 the expression $e_1\ m\ e_2$ is treated as equivalent to the method call $e_1.m(e_2)$. The expression **`this`** `< that` in class `Ordered` is thus simply a more convenient way to express the method call **`this`**`.<(that)`.

The new cell class can be defined in a generic way using *bounded* type abstraction:

```
abstract class MaxCell extends AbsCell {
  type T <: Ordered { type O = T }
  def setMax(x: T) = if (get < x) set(x)
}
```

Here, the type declaration of `T` is constrained by an upper type bound which consists of a class name `Ordered` and a refinement `{ type O = T }`. The upper bound restricts the specializations of `T` in subclasses to those subtypes τ of `Ordered` for which the type member `O` of τ equals `T`.

Because of this constraint, the "<" method of class `Ordered` is guaranteed to be applicable to a receiver and an argument of type `T`. The example shows that the bounded type member may itself appear as part of the bound, i.e. Scala supports *F-bounded polymorphism* [6].

2.2 Modular Mixin Composition

After having explained Scala's constructs for type abstraction, we now focus on its constructs for class composition. Mixin class composition in Scala is a fusion of the object-oriented, linear mixin composition of Bracha [3], and the more symmetric approaches of mixin modules [8, 18] and traits [41]. To start with an example, consider the following abstraction for iterators.

```
trait AbsIterator {
  type T;
  def hasNext: boolean;
  def next: T;
}
```

Note the use of the keyword **trait** instead of **class**. A *trait* is a special form of an abstract class which does not have any value parameters for its constructor. Traits can be used in all contexts where other abstract classes appear; however only traits can be used as mixins (see below).

The `Iterator` trait is written using an abstract type member `T` which represents the iterator's element type. One could alternatively have chosen a generic representation – in fact that's what is done in the Scala standard library.

Next, consider a trait which extends `Iterator` with a method `foreach`, which applies a given function to every element returned by the iterator.

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

The parameter `f` has type `T => unit`, i.e. it is a function that takes arguments of type `T` and returns results of the trivial type `unit`.

Here is a concrete iterator class, which returns successive 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 = i + 1; x }
}
```

We now would like to combine the functionality of `RichIterator` and `StringIterator` in a single class. With single inheritance and interfaces alone this is impossible, as both classes contain member implementations with code. Therefore, Scala provides a *mixin-class composition* mechanism which allows programmers to reuse the delta of a class definition, i.e., all new definitions that are not inherited. This mechanism makes it possible to combine `RichIterator` with `StringIterator`, as is done in the following test program. The program prints a column of all the characters of a given string.

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

The `Iter` class in function `main` is constructed from a mixin composition of the parents `StringIterator` and `RichIterator`. The first parent is called the *superclass* of `Iter`, whereas the second parent is called a *mixin*.

Class Linearization

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 in general a directed acyclic graph. A linearization of this 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:

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

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

$$\begin{array}{rcll} \{a, A\} \vec{+} B & = & a, (A \vec{+} B) & \textbf{if} a \notin B \\ & = & A \vec{+} B & \textbf{if} a \in B \end{array}$$

For instance, the linearization of class `Iter` is

```
{ Iter, RichIterator, StringIterator,
  AbsIterator, AnyRef, Any }
```

The linearization of a class refines the inheritance relation: if C is a subclass of D, then C precedes D in any linearization where both C and D occur. Definition 2.1 also satisfies the property that a linearization of a class always contains the linearization of its direct superclass as a suffix. For instance, the linearization of `StringIterator` is

```
{ StringIterator, AbsIterator, AnyRef, Any } ,
```

which is a suffix of the linearization of its subclass `Iter`. The same is not true for the linearization of mixin classes. It is also possible that classes of the linearization of a mixin class appear in different order in the linearization of an inheriting class, i.e. linearization in Scala is not monotonic [2].

Membership

The `Iter` class inherits members from both `StringIterator` and `RichIterator`. Generally, a class derived from a mixin composition C_n **with** ... **with** C_1 can define members itself and can inherit members from all parent classes. Scala adopts Java and C#'s conventions for static overloading of methods. It is thus possible that a class defines and/or inherits several methods with the same name[1]. To decide whether a defined member of a class C overrides a member of a parent class, or whether the two co-exist as overloaded variants in C, Scala uses the following definition of *matching* on members, which is derived from similar concepts in Java and C#:

Definition 2.2 A member definition M *matches* a member definition M', if M and M' bind the same name, and one of following holds.

1. Neither M nor M' is a method definition.
2. M and M' define both monomorphic methods with equal argument types.
3. M and M' define both polymorphic methods with equal number of argument types $\overline{T}$, $\overline{T}'$ and equal numbers of type parameters $\bar{t}$, $\bar{t}'$, say, and $\overline{T}' = [\bar{t}'/\bar{t}]\overline{T}$.

Member definitions of a class fall into two categories: concrete and abstract. There are two rules that determine the set of members of a class, one for each category:

Definition 2.3 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 also determines the overriding relationships between matching members of a class C and its parents. 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' id defined.

Super calls

Consider the following class of synchronized iterators, which ensures that its operations are executed in a mutually exclusive way when called concurrently from several threads.

```
abstract class SyncIterator extends AbsIterator {
  abstract override def hasNext: boolean =
    synchronized(super.hasNext);
  abstract override def next: T =
    synchronized(super.next);
}
```

[1]One might disagree with this design choice because of its complexity, but it is necessary to ensure interoperability, for instance when inheriting from Java's Swing libraries.

To obtain rich, synchronized iterators over strings, one uses a mixin composition involving three classes:

```
StringIterator(someString) with RichIterator
                           with SyncIterator
```

This composition inherits the two members `hasNext` and `next` from the mixin class `SyncIterator`. Each method wraps a `synchronized` application around a call to the corresponding member of its superclass.

Because `RichIterator` and `StringIterator` define different sets of members, the order in which they appear in a mixin composition does not matter. In the example above, we could have equivalently written

```
StringIterator(someString) with SyncIterator
                           with RichIterator
```

There's a subtlety, however. The class accessed by the **super** calls in `SyncIterator` is not its statically declared superclass `AbsIterator`. This would not make sense, as `hasNext` and `next` are abstract in this class. Instead, *super* accesses the superclass `StringIterator` of the mixin composition in which `SyncIterator` takes part. In a sense, the superclass in a mixin composition *overrides* the statically declared superclasses of its mixins. It follows that calls to *super* cannot be statically resolved when a class is defined; their resolution has to be deferred to the point where a class is instantiated or inherited. This is made precise be the following definition.

Definition 2.4 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.

Note finally that in a language like Java or C#, the *super* calls in class `SyncIterator` would be illegal, precisely because they designate abstract members of the static superclass. As we have seen, Scala allows this construction, but it still has to make sure that the class is only used in a context where *super* calls access members that are concretely defined. This is enforced by the occurrence of the **abstract** and **override** modifiers in class `SyncIterator`. An **abstract override** modifier pair in a method definition indicates that the method's definition is not yet complete because it overrides and uses an abstract member in a superclass. A class with incomplete members must be declared abstract itself, and subclasses of it can be instantiated only once all members overridden by such incomplete members have been redefined.

Calls to *super* may be threaded so that they follow the class linearization (this is a major difference between Scala's mixin composition and multiple inheritance schemes). For example, consider another class similar to `SyncIterator` which prints all returned elements on standard output.

```
abstract class LoggedIterator extends AbsIterator {
  abstract override def next: T = {
    val x = super.next; System.out.println(x); x
  }
}
```

One can combine synchronized with logged iterators in a mixin composition:

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

The linearization of `Iter2` is

```
{ Iter2, LoggedIterator, SyncIterator,
  StringIterator, AbsIterator, AnyRef, Any }
```

Therefore, class `Iter2` inherits its `next` method from class `LoggedIterator`, the **`super`**`.next` call in this method refers to the `next` method in class `SyncIterator`, whose **`super`**`.next` call finally refers to the `next` method in class `StringIterator`.

If logging should be included in the synchronization, this can be achieved by reversing the order of the mixins:

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

In either case, calls to `next` follow via *super* the linearization of class `Iter2`.

2.3 Selftype Annotations

Each of the operands of a mixin composition C_0 **`with ... with`** C_n, must refer to a class. The mixin composition mechanism does not allow any C_i to refer to an abstract type. This restriction makes it possible to statically check for ambiguities and override conflicts at the point where a class is composed. Scala's selftype annotations provide an alternative way of associating a class with an abstract type. The following example illustrates this for a generic implementation of directed graphs that abstracts over its concrete node type:

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

The abstract `Node` type is upper-bounded by `BaseNode` to express that we want nodes to support a `connectWith` method. This method creates a new instance of class `Edge` which links the receiver node with the argument node. Unfortunately, this code does not compile, because the type of the self reference **`this`** is `BaseNode` and therefore does not conform to type `Node` which is expected by the constructor of class `Edge`. Thus, we have to state somehow that the identity of class `BaseNode` has to be expressible as type `Node`. Here is a possible encoding:

```
abstract class Graph {
  type Node <: BaseNode;
  abstract class BaseNode {
    def connectWith(n: Node): Edge = new Edge(self, n);
    def self: Node;
  }
  class Edge(from: Node, to: Node) { ... }
}
```

This version of class `BaseNode` uses an abstract method `self` for expressing its identity as type `Node`. Concrete subclasses of `Graph` have to define a concrete `Node` class for which it is possible to implement method `self`. This is illustrated in the code for class `LabeledGraph`.

```
class LabeledGraph extends Graph {
  class Node(label: String) extends BaseNode {
    def getLabel: String = label;
    def self: Node = this;
  }
}
```

This programming pattern appears quite frequently when family polymorphism is combined with explicit references to **`this`**. Therefore, Scala supports a mechanism for specifying the type of **`this`** explicitly. Such an *explicit selftype annotation* is used in the following version of class `Graph`:

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

In the declaration

```
  class BaseNode requires Node { ...
```

`Node` is called the *selftype* of class `BaseNode`. When a selftype is given, it is taken as the type of **`this`** inside the class. Without a selftype annotation, the type of **`this`** is taken as usual to be the type of the class itself. In class `BaseNode`, the selftype is necessary to render the call **`new`** `Edge(`**`this`**`, n)` type-correct.

Selftypes can be arbitrary; they need not have a relation with the class being defined. Type soundness is still guaranteed, because of two requirements: (1) the selftype of a class must be a subtype of the selftypes of all its base classes, (2) when instantiating a class in a **`new`** expression, it is checked that the selftype of the class is a supertype of the type of the object being created.

Selftypes were first introduced in the νObj calculus, mainly for technical reasons. We expected initially that they would not be used very frequently in Scala programs, but included them anyway since they seemed essential in situations where family polymorphism is combined with explicit self references. To our surprise, selftypes turned out to be the key construct for lifting static systems to component-based systems. This is further explained in Section 4.

2.4 Service-Oriented Component Model

The presented class abstraction and composition mechanisms form the basis of a *service-oriented software component model*. Software components are units of computation that *provide* a well-defined set of services. Typically, a software component is not self-contained; i.e., its service implementations rely on a set of *required services* provided by other cooperating components.

In our model, software components correspond to classes. Concrete members of a class represent provided services, whereas abstract members represent required services. Component composition is based on mixins, which lets one

create bigger components from smaller ones.

The mixin-class composition mechanism identifies services with the same name; for instance, an abstract method m can be implemented by a class C defining a concrete method m simply by mixing-in C. Thus, the component composition mechanism automatically associates required with provided services. Together with the rule that concrete class members always override abstract ones, this principle yields recursively pluggable components where component services do not have to be wired explicitly [50].

This approach simplifies the assembly of large components with many recursive dependencies. It scales well even in the presence of many required and provided services, since the association of the two is automatically inferred by the compiler. The most important advantage over traditional black-box components is that components are extensible entities: they can evolve by subclassing and overriding. They can even be used to add new services to other existing components, or to upgrade existing services of other components. Overall, these features enable a smooth incremental software evolution process [52].

3. CASE STUDY: SUBJECT/OBSERVER

The abstract type concept is particularly well suited for modeling families of types which vary together covariantly. This concept has been called *family polymorphism* [11]. As an example, consider the publish/subscribe design pattern. There are two classes of participants – subjects and observers. Subjects define a method `subscribe` by which observers register. They also define a `publish` method which notifies all registered observers. Notification is done by calling a method `notify` which is defined by all observers. Typically, `publish` is called when the state of a subject changes. There can be several observers associated with a subject, and an observer might observe several subjects. The `subscribe` method takes the identity of the registering observer as parameter, whereas an observer's `notify` method takes the subject that did the notification as parameter. Hence, subjects and observers refer to each other in their method signatures.

All elements of the subject/observer design pattern are captured in the following system.

```
abstract class SubjectObserver {
  type S <: Subject;
  type O <: Observer;
  abstract class Subject requires S {
    private var observers: List[O] = List();
    def subscribe(obs: O) =
      observers = obs :: observers;
    def publish =
      for (val obs <- observers) obs.notify(this);
  }
  abstract class Observer {
    def notify(sub: S): unit;
  }
}
```

The top-level class `SubjectObserver` has two member classes: one for subjects, the other for observers. The `Subject` class defines methods `subscribe` and `publish`. It maintains a list of all registered observers in the private variable `observers`. The `Observer` class only declares an abstract method `notify`.

Note that the `Subject` and `Observer` classes do not directly refer to each other, since such "hard" references would prevent covariant extensions of these classes in client code. Instead, `SubjectObserver` defines two abstract types `S` and `O` which are bounded by the respective class types `Subject` and `Observer`. The subject and observer classes use these abstract types to refer to each other.

Note also that class `Subject` relies on an explicit selftype annotation, which is necessary to render the method call `obs.notify(this)` type-correct.

The mechanism defined in the publish/subscribe pattern can be used by inheriting from `SubjectObserver`, defining application specific `Subject` and `Observer` classes. An example is the `SensorReader` object, which defines sensors as subjects and displays as observers.

```
object SensorReader extends SubjectObserver {
  type S = Sensor;
  type O = Display;
  abstract class Sensor extends Subject {
    val label: String;
    var value: double = 0.0;
    def changeValue(v: double) = {
      value = v;
      publish;
    }
  }
  class Display extends Observer {
    def println(s: String) = ...
    def notify(sub: Sensor) =
      println(sub.label + "␣has␣value␣" + sub.value);
  }
}
```

An object definition such as the one for `SensorReader` creates a *singleton class* which has as a single instance the defined object. In the `SensorReader` object, type `S` is bound to `Sensor` whereas type `O` is bound to `Display`. Hence, the two formerly abstract types are now defined by overriding definitions. This "tying the knot" is always necessary when creating a concrete class instance. On the other hand, it would also have been possible to define an abstract `SensorReader` class which could be refined further by client code. In this case, the two abstract types would have been overridden again by abstract type definitions.

```
abstract class AbsSensorReader extends SubjectObserver {
  type S <: Sensor;
  type O <: Display;
  ...
}
```

The following program illustrates how the `SensorReader` object is used.

```
object Test {
  import SensorReader._;
  val s1 = new Sensor { val label = "sensor1" }
  val s2 = new Sensor { val label = "sensor2" }
  def main(args: Array[String]) = {
    val d1 = new Display; val d2 = new Display;
    s1.subscribe(d1); s1.subscribe(d2);
    s2.subscribe(d1);
    s1.changeValue(2); s2.changeValue(3);
  }
}
```

Note the presence of an `import` clause, which makes the members of object `SensorReader` available without prefix to the code in object `Test`. Import clauses in Scala are more general than import clauses in Java. They can be used anywhere, and can import members from any object, not just from a package.

The Subject/Observer pattern has been studied by several groups before. A solution structurally close to ours but based on virtual types has been sketched by Thorup [44]. The development in this section shows by example that Beta's virtual types can be emulated by a combination of Scala's abstract types and explicitly typed self references. Other approaches to expressing the publish/subscribe pattern are based on a generalization of *mytype* [4] or on parametric polymorphism using OCaml's row-variables to model extensible records [40].

4. CASE STUDY: THE SCALA COMPILER

The Scala compiler, *scalac*, consists of several phases. The first phase is syntax analysis, implemented by a scanner and a conventional recursive descent parser. The result of this phase is an abstract syntax tree. The next phase attributes the syntax tree with symbol and type information. This is followed by a number of phases that transform the syntax tree. Most transformations replace some high-level Scala-specific constructs with lower-level constructs that can more directly be represented in bytecode. Other transformations perform optimizations such as inlining or tail call elimination. Transformations always consume and produce attributed trees.

All phases after syntax analysis work with a symbol table. This table itself consists of a number of modules. Some of these are:

- A module `Names` that represents symbol names. A name is represented as an object consisting of an index and a length, where the index refers to a global array in which all characters of all names are stored. A hashmap ensures that names are unique, i.e. that equal names always are represented by the same object.
- A module `Symbols` that represents symbols corresponding to definitions of entities like classes, methods, variables, etc. in Scala and Java modules.
- A module `Types` that represents types.
- A module `Definitions` that contains globally visible symbols for definitions that have a special significance for the Scala compiler. Examples are Scala's value classes, the top and bottom classes `scala.Any` and `scala.All`, or the boolean values **`true`** and **`false`**.
- A module `Scopes` that represents local scopes and class sets of class members.

The structure of these modules is highly recursive. For instance, every symbol has a type, and some types also have a symbol. The `Definitions` module creates symbols and types, and is in turn used by certain operations in `Types`. References between modules involve member accesses, object creations, but also inheritance. For instance, the types of many symbols are lazily created, so that forward references in definitions can be supported and library class and source files can be loaded on demand. This is achieved by initializing the types of symbols to special "lazy types" that replace themselves with a symbol's true type the first time the symbol is accessed. Lazy types deal with the dynamics of compilation instead of the type structure; consequently, they are defined outside the `Types` module, even though they inherit from the `Type` class.

State of the art

In previously released versions of the Scala compiler, all modules described above were implemented as top-level classes (implemented in Java), which contain static members and data. For instance, the contents of names were stored in a static array in the `Names` class. Likewise, global symbols were stored as static data in the `Definitions` class. This technique has the advantage that it supports complex recursive references. But it also has two disadvantages. First, since all references between classes were hard links, we could not treat compiler classes as components that can be combined with different other components. This, in effect, prevented piecewise extensions or adaptations of the compiler. Second, since the compiler worked with mutable static data structures, it was not re-entrant, i.e. it was not possible to have several concurrent executions of the compiler in a single VM. This was a problem for using the Scala compiler in an integrated development environment such as Eclipse.

These problems are of course not new. For instance, the Java compilers *javac* and *JaCo* [53] have a structure similar to the one of *scalac*. In these compilers, static data structures and static component references are avoided by using a design pattern which parameterizes compiler components with a *context*. A context is a mapping from component identifiers to component implementations (objects). A compiler component uses the context to get access to cooperating runtime components.

This approach makes it possible to run several compilers in one VM simply by creating different contexts with independent instantiations of the compiler components. On the other hand, there are several disadvantages. First of all, a simple solution, like the one used in *javac*, models contexts as maps from names to objects. This approach is subject to dynamic typing and thus statically unsafe. *JaCo*'s *Context/Component* design pattern uses a combination of an object repository and an abstract factory to model contexts [49, 52]. This pattern provides static type safety, but is associated with a relatively high protocol overhead. For instance, *JaCo*'s 30000 lines of code include 600 lines of code just for context definitions and more than 1200 lines of code for object factories, not counting the code within the actual compiler components that use the contexts and the factories. Contexts also break encapsulation because they require that data structures are packaged outside the classes that access them.

Beyond the protocol overhead, static typing, and encapsulation issues there is always the risk to violate the programming pattern, since there is no way to enforce the design statically. For instance, if two instances of a compiler are executed simultaneously, and one name table is allocated per compiler run, it becomes important that names referring to different compiler instances are kept distinct. Otherwise a name might index a table which does not store its characters but some random characters. This isolation cannot be guaranteed statically.

```
class SymbolTable {
  class Name { ... }
  // name specific operations

  class Type { ... }
  // subclasses of Type and type specific operations

  class Symbol { ... }
  // subclasses of Symbol and symbol specific operations

  object definitions { // global definitions }

  // other elements
}
```

Listing 1: *scalac*'s symbol table structure

Another solution to the problem is to use programming languages providing constructs for component composition and abstraction. For instance, functors of the SML module system [27] can be used to implement component-based systems where component interactions are not hard-coded. On the other hand, functors are neither first-class nor higher-order. Consequently, they cannot be used to create new compilers from dynamically provided components. Other module systems, like MzScheme's *Units* [15, 14], are expressive enough to allow this, but they are often only dynamically typed, giving no guarantees at compile-time. Typical component-oriented programming languages like ArchJava [1], Jiazzi [30], and ComponentJ [42] are statically typed and do provide good support for creating and composing generic software components, but their type systems are not expressive enough to fully isolate reentrant systems. The module system of Keris [51] can enforce a strict separation of multiple reentrant instances of a compiler, but without support for first-class modules it requires that the number of simultaneously running compiler instances is known statically.

A simple reentrant compiler implementation

For the rewrite of the Scala compiler we found another solution, which is type safe, and which uses the language elements of Scala itself. As a first step towards this solution, we introduce nesting of classes to express local structure. A simplified version of the symbol table component of the *scalac* compiler – to be refined later – is shown in Listing 1.

Here, classes `Name`, `Symbol`, `Type`, and the object `Definitions` are all members of the `SymbolTable` class. The whole compiler (which would be structured similarly) can access definitions in this class by inheriting from it:

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

In that way, we arrive at a compiler without static definitions. The compiler is by design re-entrant, and can be instantiated like any other class as often as desired. Furthermore, member types of different instantiations are isolated from each other, which gives a good degree of type safety. Consider for instance a scenario where two instances `c1` and `c2` of the Scala compiler co-exist.

```
abstract class Types requires (Types with Names
                                  with Symbols
                                  with Definitions) {
  class Type { ... }
  // subclasses of Type and
  // type specific operations
}
abstract class Symbols requires (Symbols with Names
                                    with Types) {
  class Symbol { ... }
  // subclasses of Symbol and
  // symbol specific operations
}
abstract class Definitions requires
                          (Definitions with Names
                                       with Symbols){
  object definitions { ... }
}
abstract class Names {
  class Name { ... }
  // name specific operations
}
class SymbolTable extends Names
                    with Types
                    with Symbols
                    with Definitions;

class ScalaCompiler extends SymbolTable
                      with Trees
                      with ... ;
```

Listing 2: Symbol table components with required interfaces

```
val c1 = new ScalaCompiler;
val c2 = new ScalaCompiler;
```

Names created by the `c1` compiler instance have the path-dependent type `c1.Name`, whereas names created by `c2` have type `c2.Name`. Since these two types are incompatible, a problematic assignment such as the following would be ruled out.

```
c1.definitions.AllClass.name =
  c2.definitions.AllClass.name // illegal!
```

Component-based implementation

The code sketched above has a very severe shortcoming: it is a large monolithic program and thus not really component-based! Indeed, the whole symbol table code (roughly 4000 lines) is now placed in a single source file. This clearly becomes impractical for large programs.

Nevertheless, the previous attempt points the way to a solution. We need to express a nested structure like the one above, but with its constituents spread over separate source files. The problem is how to express cross-file references in this setting. For instance, in class `Symbol` one needs to refer to the corresponding `Type` class which belongs to the same compiler instance but which is defined in a different source file.

There are several possible solutions to this problem. The solution we have chosen is sketched in Listing 2. It uses an

explicit selftype to express the required services of a component.

The `Types` class contains a class hierarchy rooted in class `Type` as well as operations that relate to types. It comes with an explicit selftype, which is an intersection type of all classes required by `Types`. Besides `Types` itself, these classes are `Names`, `Symbols`, and `Definitions`. Members of these classes are thus accessible in class `Types`. For instance, one can write `this.Symbol` or shorter just `Symbol` for the `Symbol` class member of the required `Symbols` class.

The schema for the other symbol table classes follows the one for types. In each case, all required classes are listed as operands of an intersection type in an explicit selftype annotation. The whole symbol table class is then simply the mixin composition of these components. Figure 2 illustrates this principle. For every component, it shows the provided classes as well as the classes that are required from other components. Classes are represented by boxes, object definitions are represented by ovals. Combining all components via mixin composition yields a fully self-contained component without any required classes. This class represents our complete instantiatable symbol table abstraction.

The presented scheme is statically type safe, and provides explicit notation to express required as well as provided interfaces of a component. It is concise, since no explicit wiring, for example by means of parameter passing, is necessary. It provides great flexibility for component structuring. In fact it allows to lift arbitrary module structures with static data and hard references to component systems.

Variants

Granularity of dependency specifications.

The presented scheme is not the only possible solution. Several variants are possible, which differ in the way required components are abstracted. For instance, one can be more concise but less precise in assuming as selftype of each symbol table component the `SymbolTable` class itself. E.g.:

```
class Types requires SymbolTable { ... }
```

One can also characterize required services in more detail by using abstract type and value members. E.g:

```
class Types {

  type Symbol <: SymbolInterface;
  type Name <: NameInterface;
  // other required types

  def newValue(name: Name): Symbol;
  // other required values

  class Type { ... }
  ...
}
```

One can thus narrow required services to arbitrary sets of component members, whereas previously one could require components only as a whole. The price to be paid for the precision is a loss of conciseness, since bounds of abstract types such as `SymbolInterface` in the code above have to be defined explicitly. Furthermore, abstracted types cannot be inherited, since abstract types in Scala cannot be superclasses or mixins.

Hierarchical organization of components.

In all variations, the symbol table class itself results from a mixin composition of all its constituent classes. From a system view, all symbol table components are defined on the same level. But it is also possible to define subsystems which can be nested in other components by means of aggregation. An example is the parser phase component of *scalac*:

```
class ParserPhase extends Lexical with Syntactic {
  val compiler: Compiler;
}
```

Here, the sub-components `Lexical` and `Syntactic` are structured similarly to the symbol table components with self types expressing required components. The syntactic analysis phase also needs to access the compiler as a whole, for instance for reporting errors or for constructing syntax trees. These accesses are done via a member field `compiler`, which is abstract in class `ParserPhase`. The corresponding integration of the parser phase object in the *scalac* compiler is sketched in the listing below.

```
class ScalaCompiler extends SymbolTable with Trees {
  object parserPhase extends ParserPhase {
    val compiler: ScalaCompiler.this.type =
      ScalaCompiler.this
  }
  ...
}
```

Class `ScalaCompiler` defines an instance of class `ParserPhase` in which the `compiler` field is bound to the enclosing `ScalaCompiler` instance itself. The type of that field is the singleton type `ScalaCompiler.this.type`, which has as the only member the current instance of `ScalaCompiler`. The singleton type annotation is necessary since `ParserPhase` contains members that refer to types defined in `ScalaCompiler`. An example is the type `Tree` of abstract syntax trees, which `ScalaCompiler` inherits from class `Trees`. To connect the tree generated by the parser phase with later phases, the type checker needs to know the type equality

```
parserPhase.compiler.Tree  =  Tree
```

in the context of `ScalaCompiler.this`. The singleton type annotation establishes `ScalaCompiler.this` as an alias of `ScalaCompiler.this.parserPhase.compiler` and therefore validates the above equality.

Component adaptation

The new compiler architecture makes adaptations very easy. As an example, consider logging. Let's say we want to log every creation of a symbol or a type in the Scala compiler. Logging involves writing information on some output channel `log`, of type `java.io.PrintStream`. The crucial point is that we want to extend an existing compiler with logging functionality. To do this, we do not want to modify the compiler's source code. Neither do we want to require of the compiler writer to have pre-planned the logging extension by providing hooks. Such hooks tend to impair the clarity of the code since they mix separate concerns in one class. Instead, we use subclassing to add logging functionality to existing classes. E.g.:

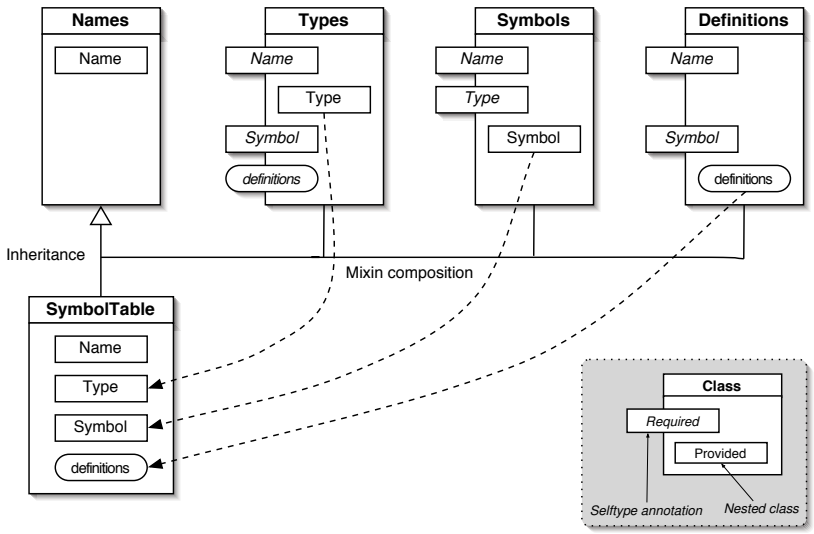

Figure 2: Composition of the Scala compiler's symbol tables.

```
abstract class LogSymbols extends Symbols {
  val log: java.io.PrintStream;
  override def newTermSymbol(name: Name): TermSymbol =
  {
    val x = super.newTermSymbol(name);
    log.println("creating term symbol " + name);
    x
  }
  // similarly for all other symbol creations.
}
```

Analogously, one can define a subclass `LogTypes` of class `Types` to log all type creations.

The question then is how to inject the logging behavior into an existing system. Since the whole Scala compiler is defined as a single class, this is a straightforward application of mixin composition:

```
class LoggedCompiler extends ScalaCompiler
                     with LogSymbols with LogTypes {
  val log: PrintStream = System.out
}
```

In the mixin composition the new implementation of `newTermSymbol` in class `LogSymbols` overwrites the implementation of the same method which is defined in class `Symbol` and which is inherited by class `ScalaCompiler`. Conversely, the abstract members named `log` in classes `LogSymbols` and `LogTypes` are replaced by the concrete definition of `log` in class `LoggedCompiler`.

This adaptation might seem trivial. But note that in a classical system architecture with static components and hard links, it would have been impossible. For such architectures, aspect-oriented programming [22] proposes an alternative solution, which is based on code rewriting. In fact, our component architecture can handle some of the scenarios for which AOP has been proposed as the technique of choice. Other examples besides logging are synchronization, security checking, or choice of data representation. More generally, our architecture can handle all *before*, *after*, and *around* advice on method reception pointcut designators. These represent only one instance of the pointcut designators provided by languages such as AspectJ [21]. Therefore, general AOP is clearly more powerful than our scheme. On the other hand, our scheme has the advantage that it is statically typed, and that scope and order of advice can be precisely controlled using the semantics of mixin composition.

5. DISCUSSION

We have identified three building blocks for the construction of reusable components: abstract type members, explicit selftypes, and symmetric mixin composition. The three building blocks were formalized in the νObj calculus and were implemented in Scala. Scala is also the language in which all programming examples and case studies of this paper are written. It constitutes thus a concrete experiment which validates the construction principles presented here in a range of applications written by many different people.

But Scala is, of course, not the only possible language design that would enable such constructions. In this section, we try to generalize from Scala's concrete setting, in order to identify what language constructs are essential to achieve systems of scalable and dynamic components. We assume in the whole discussion a strongly and statically typed object-oriented language. The situation is quite different for dynamically typed languages, and is different again for functional languages with ML-like module systems.

The first important language construct is class nesting. Since class nesting is already supported by mainstream languages, we have omitted it from our discussion so far, but it is essential nonetheless. It is the primary means for aggregation and encapsulation. Without it, we could only compose systems consisting of fields and methods, but not systems that contain themselves classes. Said otherwise, every class would have to be either a base-class or mixin of a top-level system (in which case it would only have one instance per top-level instantiation), or it would be completely external to that system (in which case it cannot access anything hid-

den in the system). It would still be possible to construct component-based systems as discussed by this paper, but the necessary amount of wiring would be substantial, and one would have to give up object-oriented encapsulation principles to a large extent.

The second language construct is some form of mixin or trait composition or multiple inheritance. Not all details have to be necessarily done the way they were done in Scala's symmetric mixin composition. We only require two fundamental properties: First, that mixins or classes can contain themselves mixins or classes as members. Second, that concrete implementations in one mixin or class may replace abstract declarations in another mixin or class, independent of the order in which the mixins were composed. The latter property is necessary to implement mutually recursive dependencies between components.

The third language construct is some means of abstraction over the required services of a class. Such abstraction has to apply to all forms of definitions that can occur inside a class. In particular it must be possible to abstract over classes as well as methods. We have seen in Scala two means of abstraction. One worked by abstracting over class members, the other by abstracting over the type of self. These two techniques are largely complementary in what they achieve.

Abstraction over class members gives very fine-grained control over required types and services. Each required entity is named individually, and also can be given a type (or type-bound in the case of type members) which captures only what is required from the entity by the containing class. The entity may then be defined in another class with a stronger type (or type-bound) than the required one. In other words, class member abstraction introduces "type-slack" between the required and provided interfaces for the same service. This in turn allows us to specify the required interface of a class with great precision.

Abstraction over class members also supports covariant specialization. In fact, this is a consequence of the type-slack it introduces. Covariant specialization is important in many different situations. One set of situations is characterized by the generic "expression problem" example. Here, the task is to extend systems over a recursive data type by new data variants as well as by new operations over that data [45, 37]. Related to this is also the production line problem where a set of features has to be composed in a modular way to yield a software product [26]. Family polymorphism is another instance of covariant specialization. Here, several types need to be specialized together, as in the subject/observer example of Section 3.

The downside of the precision of class member abstraction is its verbosity. Listing all required methods, fields, and types including their types and type bounds can add significant overhead to a component's description. Selftype abstraction is a more concise alternative to member abstraction. Instead of naming and typing all members individually one simply attaches a type to **this**. This is somewhat akin to the difference between structural and nominal typing.

In fact, selftype abstractions are almost as concise as traditional references between static components. To see this, note that import clauses in traditional systems correspond to summands in a compound selftype in our scheme. Consider for instance a system of three Java classes `A`, `B`, and `C`, each of which refers to the other two. Assume that all three classes contain static nested classes. Then class `A` could import all nested classes in `B` and `C` using code like this:

```
import B.*;
import C.*;
class A { ... }
```

Classes `B` and `C` would be organized similarly.

Translating Java's static setting into one where components can be instantiated multiple times, we obtain the following, slightly more concise Scala code:

```
class A requires (A with B with C) { ... }
```

Classes `B` and `C` are organized similarly. The inter-class references in `A`, `B`, and `C` stay exactly the same. In particular, all nested classes can be accessed without qualification. The only piece of code that needs to be written in addition is a definition of a top-level application which contains all three classes:

```
class All extends A with B with C;
```

In the case of static components, the definition of the set of classes making up an application is implicit — it is the transitive closure of all classes reachable from the main program.

In Scala, there is a second advantage of selftype abstraction over class member abstraction. This has to do with a shortcoming of class member abstraction as it is defined in the language. In fact, Scala allows member abstraction only over types, but lacks the possibility to abstract over other aspects of classes. Abstract types can be used as types for members, but no instances can be created from them, nor can they be inherited by subclasses. Hence, if some of the classes defined in a component inherit from some external class in the component's required interface, selftype abstraction is the only available means to express this. The same holds if a component instantiates objects from an external, required class using **new** rather than going through a factory method.

Lifting the restrictions on class member abstraction would lead us from abstract types to virtual classes in their full generality, in the way they are defined in gbeta [10], for example. This would yield a more expressive language for flexible component architectures [12]. On the other hand, the resulting language would have to either avoid or detect accidental override conflicts between pairs of classes that do not statically inherit from each other. Neither is easy to type-check or to implement on standard platforms such as JVM or the .NET CLR.

6. CONCLUSION

We have presented three building blocks for reusable components: abstract type members, explicit selftypes, and modular mixin composition. Each of these constructs exists in some form also in other formalisms, but we believe to be the first to combine them in one language and to have discovered the importance of their combination in building and composing software components. We have demonstrated their use in two case studies, a publish/subscribe framework and the Scala compiler itself. The case studies show that our language constructs are adequate to lift an arbitrary assembly of static program parts to a component system where required interfaces are made explicit and hard links between components are avoided. The lifting completely

preserves the structure of the original program.

This is not the end of the story, however. The scenario we have studied was the initial construction of a statically typed system of components running on a single site. We did not touch aspects of distribution and dynamic component discovery, nor did we treat the evolution of a component system over time. We intend to focus on these topics in future work.

Acknowledgments. The Scala design and implementation has been a collective effort of many people. Besides the authors, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, and Erik Stenman have made important contributions. The work was partially supported by grants from the Swiss National Fund under project NFS 21-61825, the Swiss National Competence Center for Research MICS, Microsoft Research, and the Hasler Foundation. We also thank Gilad Bracha, Stéphane Ducasse, Erik Ernst, Nastaran Fatemi, Matthias Felleisen, Shriram Krishnamurti, Oscar Nierstrasz, Didier Rémy, and Philip Wadler for useful discussions about the material presented in this paper.

7. REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.

[2] K. Barrett, B. Cassels, P. Haahr, D. A. Moon, K. Playford, and P. T. Withington. A monotonic superclass linearization for dylan. In *Proc. OOPSLA*, pages 69–82. ACM Press, Oct. 1996.

[3] G. Bracha and W. Cook. Mixin-Based Inheritance. In N. Meyrowitz, editor, *Proceedings of* ECOOP '90, pages 303–311, Ottawa, Canada, October 1990. ACM Press.

[4] K. B. Bruce, M. Odersky, and P. Wadler. A Statically Safe Alternative to Virtual Types. *Lecture Notes in Computer Science*, 1445, 1998. Proc. ESOP 1998.

[5] K. B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language. In *Proceedings of* ECOOP '95, LNCS 952, pages 27–51, Aarhus, Denmark, August 1995. Springer-Verlag.

[6] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-Bounded Quantification for Object-Oriented Programming. In *Proc. of 4th Int. Conf. on Functional Programming and Computer Architecture, FPCA'89, London*, pages 273–280, New York, Sep 1989. ACM Pres.

[7] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An Extension of System F with Subtyping. *Information and Computation*, 109(1–2):4–56, 1994.

[8] D. Duggan. Mixin modules. In *ACM SIGPLAN International Conference on Functional Programming*, 1996.

[9] ECMA. C# Language Specification. Technical Report Standard ECMA-334, 2nd Edition, European Computer Manufacturers Association, December 2002.

[10] E. Ernst. *gBeta: A language with virtual attributes, block structure and propagating, dynamic inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.

[11] E. Ernst. Family polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 303–326, Budapest, Hungary, 2001.

[12] E. Ernst. Higher-Order Hierarchies. In L. Cardelli, editor, *Proceedings ECOOP 2003*, LNCS 2743, pages 303–329, Heidelberg, Germany, July 2003. Springer-Verlag.

[13] K. Fisher and J. H. Reppy. The Design of a Class Mechanism for Moby. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 37–49, 1999.

[14] M. Flatt. *Programming Languages for Reusable Software Components*. PhD thesis, Rice University, Department of Computer Science, June 1999.

[15] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[16] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Java Series, Sun Microsystems, second edition, 2000.

[17] R. Harper and M. Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, January 1994.

[18] T. Hirschowitz and X. Leroy. Mixin Modules in a Call-by-Value Setting. In *European Symposium on Programming*, pages 6–20, 2002.

[19] A. Igarashi and M. Viroli. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. In *Proceedings of the Sixteenth European Conference on Object-Oriented Programming (ECOOP2002)*, pages 441–469, June 2002.

[20] M. P. Jones. Using parameterized signatures to express modular structure. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 68–78. ACM Press, 1996.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of* ECOOP 2001, Springer LNCS, pages 327–353, 2001.

[22] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, Jyväskylä, Finland, 1997.

[23] J. L. Knudsen. Aspect-oriented programming in beta using the fragment system. In *Proceedings of the Workshop on Object-Oriented Technology*, Springer LNCS, pages 304–305, 1999.

[24] X. Leroy. Manifest Types, Modules and Separate Compilation. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 109–122, January 1994.

[25] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.00, documentation and user's manual, April 2000.

[26] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. In *Proceedings of the European Conference on Object-Oriented Programming*, number July in Springer LNCS, 2005.

[27] D. MacQueen. Modules for Standard ML. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Papers Presented at the Symposium, August 6–8, 1984*, pages 198–207, New York, August 1984. Association for Computing Machinery.

[28] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. ddison Wesley, June 1993.

[29] O. L. Madsen and B. Moeller-Pedersen. Virtual Classes - A Powerful Mechanism for Object-Oriented Programming. In *Proc. OOPSLA'89*, pages 397–406, October 1989.

[30] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age Components for Old-Fashioned Java. In *Proc. of OOPSLA*, October 2001.

[31] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings of OOPSLA '02, Sigplan Notices, 37 (11)*, pages 52–67, 2002.

[32] N. Nystrom, S. Chong, and A. Myers. Scalable Extensibility via Nested Inheritance. In *Proc. OOPSLA*, Oct 2004.

[33] Object Technology International. *Eclipse Platform Technical Overview*, Feb. 2003. www.eclipse.org.
[34] M. Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.
[35] 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.
[36] M. Odersky, C. Zenger, and M. Zenger. Colored Local Type Inference. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, pages 41–53, London, UK, January 2001.
[37] M. Odersky and M. Zenger. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, Jan. 2005. http://homepages.inf.ed.ac.uk/wadler/fool.
[38] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, Malaga, Spain, 2002.
[39] B. C. Pierce and D. N. Turner. Local Type Inference. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, pages 252–265, New York, NY, 1998.
[40] D. Rémy and J. Vuillon. On the (un)reality of virtual types. available from http://pauillac.inria.fr/remy/work/virtual, Mar. 2000.
[41] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable Units of Behavior. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, Darmstadt, Germany, June 2003.
[42] J. C. Seco and L. Caires. A basic model of typed components. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128, 2000.
[43] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley / ACM Press, New York, 1998. ISBN 0-201-17888-5.
[44] K. K. Thorup. Genericity in java with virtual types. In *Proc. ECOOP '97*, LNCS 1241, pages 444–471, June 1997.
[45] M. Torgersen. The expression problem revisited — Four new solutions using generics. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, Oslo, Norway, June 2004.
[46] M. Torgersen, E. Ernst, and C. P. Hansen. Wild FJ. In *Proc. FOOL 12*, Jan. 2005.
[47] M. Torgersen, C. P. Hansen, E. Ernst, P. vod der Ahé, G. Bracha, and N. Gafter. Adding Wildcards to the Java Programming Language. In *Proceedings SAC 2004*, Nicosia, Cyprus, March 2004.
[48] A. Wittmann. Towards Caesar: Family polymorphism for Java. Master's thesis, Technische Universität Darmstadt, Fachbereich Informatik, 2003.
[49] M. Zenger. Erweiterbare Übersetzer. Master's thesis, University of Karlsruhe, August 1998.
[50] M. Zenger. Type-Safe Prototype-Based Component Evolution. In *Proceedings of the European Conference on Object-Oriented Programming*, Málaga, Spain, June 2002.
[51] M. Zenger. Keris: Evolving software with extensible modules. To appear in *Journal of Software Maintenance and Evolution: Research and Practice (Special Issue on USE)*, 2004.
[52] M. Zenger. *Programming Language Abstractions for Extensible Software Components*. PhD thesis, Department of Computer Science, EPFL, Lausanne, March 2004.
[53] M. Zenger and M. Odersky. Implementing extensible compilers. In *ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages*, Budapest, Hungary, June 2001.

APPENDIX

A. GENERICS IN SCALA

This appendix fills in the other important part of Scala's type system, which was omitted from discussion until now. It presents the design of generics in Scala, contrasts it with the corresponding constructs in Java, and shows how generics can be encoded by abstract type members.

Scala uses a rich but fairly standard design for parametric polymorphism. Both classes and methods can have type parameters. Class type parameters can be annotated to be covariant as well as contravariant, and they can have upper as well as lower bounds.

```
class GenCell[T](init: T) {
  private var value: T = init;
  def get: T = value;
  def set(x: T): unit = { value = x }
}
def swap[T](x: GenCell[T], y: GenCell[T]): unit = {
  val t = x.get; x.set(y.get); y.set(t)
}
def main(args: Array[String]) = {
  val x: GenCell[int] = new GenCell[int](1);
  val y: GenCell[int] = new GenCell[int](2);
  swap[int](x, y)
}
```

Listing 3: Simple generic classes and methods

As a simple example, Listing 3 defines a generic class of cells of of values that can be read and written, together with a polymorphic function `swap`, which exchanges the contents of two cells, as well as a `main` function which creates two cells of integers and then swaps their contents.

Type parameters and type arguments are written in square brackets, e.g.`[T]`, `[int]`. Scala defines a sophisticated type inference system which permits to omit actual type arguments. Type arguments of a method or constructor are inferred from the expected result type and the argument types by local type inference [39, 36]. Hence, the body of function `main` in Listing 3 can also be written without any type arguments:

```
val x = new GenCell(1); val y = new GenCell(2); swap(x, y)
```

Variance

The combination of subtyping and generics in a language raises the question how they interact. If C is a type constructor and S is a subtype of T, does one also have that $C[S]$ is a subtype of $C[T]$? Type constructors with this property are called *covariant*. The type constructor `GenCell` should clearly not be covariant; otherwise one could construct the following program which leads to a type error at run time.

```
val x: GenCell[String] = new GenCell[String];
val y: GenCell[Any] = x;  // illegal!
y.set(1);
val z: String = y.get
```

It is the presence of a mutable variable in `GenCell` which makes covariance unsound. Indeed, a `GenCell[String]` is not a special instance of a `GenCell[Any]` since there are things one can do with a `GenCell[Any]` that one cannot do with a `GenCell[String]`; set it to an integer value, for instance.

On the other hand, for immutable data structures, covariance of constructors is sound and very natural. For instance, an immutable list of integers can be naturally seen as a special case of a list of `Any`. There are also cases where contravariance of parameters is desirable. An example are output channels `Chan[T]`, with a write operation that takes a parameter of the type parameter `T`. Here one would like to have `Chan[`S`] <: Chan[`T`]` whenever T `<:` S.

Scala allows to declare the variance of the type parameters of a class using plus or minus signs. A "+" in front of a parameter name indicates that the constructor is covariant in the parame-

ter, a "−" indicates that it is contravariant, and a missing prefix indicates that it is non-variant.

For instance, the following trait GenList defines a simple covariant list with methods isEmpty, head, and tail.

```
trait GenList[+T] {
  def isEmpty: boolean;
  def head: T;
  def tail: GenList[T]
}
```

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 classified as covariant for the types of immutable fields and method results, and contravariant for method argument types and upper type parameter bounds. Type arguments to a non-variant type parameter are always in non-variant position. The position flips between contra- and co-variant inside a type argument that corresponds to a contravariant parameter. The type system enforces that covariant type parameters are only used in covariant positions, and that contravariant type parameters are only used in contravariant positions.

Here are two implementations of the GenList class:

```
object Empty extends GenList[All] {
  def isEmpty: boolean = true;
  def head: All = throw new Error("Empty.head");
  def tail: List[All] = 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
}
```

As is shown in Figure 1, the type All represents the bottom type of the subtyping relation of Scala (whereas Any is the top). There are no values of this type, but the type is nevertheless useful, as shown by the definition of the empty list object, Empty. Because of co-variance, Empty's type, GenList[All] is a subtype of GenList[T], for any element type T. Hence, a single object can represent empty lists for every element type.

Binary methods and lower bounds

So far, we have associated covariance with immutable data structures. In fact, this is not quite correct, because of *binary methods*. For instance, consider adding a prepend method to the GenList trait. The most natural definition of this method takes an argument of the list element type:

```
trait GenList[+T] { ...
  def prepend(x: T): GenList[T] =   // illegal!
    new Cons(x, this)
}
```

However, this is not type-correct, since now the type parameter T appears in contravariant position inside trait GenList. Therefore, it may not be marked as covariant. This is a pity since conceptually immutable lists should be covariant in their element type. The problem can be solved by generalizing prepend using a lower bound:

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

prepend is now a polymorphic method which takes an argument of some supertype S of the list element type, T. It returns a list with elements of that supertype. The new method definition is legal for covariant lists since lower bounds are classified as covariant positions; hence the type parameter T now appears only covariantly inside trait GenList.

It is possible to combine upper and lower bounds in the declaration of a type parameter. An example is the following method less of class GenList which compares the receiver list and the argument list.

```
trait GenList[+T] { ...
  def less[S >: T <: scala.Ordered[S]](that: List[S]) =
    !that.isEmpty &&
    (this.isEmpty ||
     this.head < that.head ||
     this.head == that.head &&
       this.tail less that.tail)
}
```

The method's type parameter S is bounded from below by the list element type T and is also bounded from above by the standard class scala.Ordered[S]. The lower bound is necessary to maintain covariance of GenList. The upper bound is needed to ensure that the list elements can be compared with the < operation.

Comparison with wildcards

Java 5.0 also has a way to annotate variances which is based on wildcards [47]. The scheme is essentially a refinement of Igarashi and Viroli's variant parametric types [19]. Unlike in Scala, annotations in Java 5.0 apply to type expressions instead of type declarations. As an example, covariant generic lists could be expressed by writing every occurrence of the GenList type to match the form GenList<? extends T>. Such a type expression denotes instances of type GenList where the type argument is an arbitrary subtype of T.

Covariant wildcards can be used in every type expression; however, members where the type variable does not appear in covariant position are then "forgotten" in the type. This is necessary for maintaining type soundness. For instance, the type GenCell<? extends Number> would have just the single member get of type Number, whereas the set method, in which GenCell's type parameter occurs contravariantly, would be forgotten.

In an earlier version of Scala we also experimented with usage-site variance annotations similar to wildcards. At first-sight, this scheme is attractive because of its flexibility. A single class might have covariant as well as non-variant fragments; the user chooses between the two by placing or omitting wildcards. However, this increased flexibility comes at price, since it is now the user of a class instead of its designer who has to make sure that variance annotations are used consistently. We found that in practice it was quite difficult to achieve consistency of usage-site type annotations, so that type errors were not uncommon. This was probably partly due to the fact that we used the original system of Igarashi and Viroli [19]. Java 5.0's wildcard implementation adds to this the concept of "capture conversion" [46], which gives better typing flexibility.

By contrast, declaration-site annotations proved to be a great help in getting the design of a class right; for instance they provide excellent guidance on which methods should be generalized with lower bounds. Furthermore, Scala's mixin composition (see Section 2.2) makes it relatively easy to factor classes into covariant and non-variant fragments explicitly; in Java's single inheritance scheme with interfaces this would be admittedly much more cumbersome. For these reasons, later versions of Scala switched from usage-site to declaration-site variance annotations.

Modeling generics with abstract types

The presence of two type abstraction facilities in one language raises the question of language complexity – could we have done with just one formalism? In this section we show that functional type abstraction can indeed be modeled by object-oriented type abstraction. The idea of the encoding is as follows.

Assume you have a parameterized class C with a type parameter t (the encoding generalizes straightforwardly to multiple type parameters). The encoding has four parts, which affect the class definition itself, instance creations of the class, base class constructor calls, and type instances of the class.

1. The class definition of C is re-written as follows.

   ```
   class C {
     type t;
     /* rest of class */
   }
   ```

   That is, parameters of the original class are modeled as abstract members in the encoded class. If the type parameter t has lower and/or upper bounds, these carry over to the abstract type definition in the encoding. The variance of the type parameter does not carry over; variances influence instead the formation of types (see Point 4 below).

2. Every instance creation **new** C[T] with type argument T is rewritten to:

   ```
   new C { type t = T }
   ```

3. If C[T] appears as a superclass constructor, the inheriting class is augmented with the definition

   ```
   type t = T
   ```

4. Every type $C[T]$ is rewritten to one of the following types which each augment class C with a refinement.

`C { type t = T }`	if t is declared non-variant,
`C { type t <: T }`	if t is declared co-variant,
`C { type t >: T }`	if t is declared contra-variant.

This encoding works except for possible name-conflicts. Since the parameter name becomes a class member in the encoding, it might clash with other members, including inherited members generated from parameter names in base classes. These name conflicts can be avoided by renaming, for instance by tagging every name with a unique number.

The presence of an encoding from one style of abstraction to another is nice, since it reduces the conceptual complexity of a language. In the case of Scala, generics become simply "syntactic sugar" which can be eliminated by an encoding into abstract types. However, one could ask whether the syntactic sugar is warranted, or whether one could have done with just abstract types, arriving at a syntactically smaller language. The arguments for including generics in Scala are two-fold. First, the encoding into abstract types is not that straightforward to do by hand. Besides the loss in conciseness, there is also the problem of accidental name conflicts between abstract type names that emulate type parameters. Second, generics and abstract types usually serve distinct roles in Scala programs. Generics are typically used when one needs just type instantiation, whereas abstract types are typically used when one needs to refer to the abstract type from client code. The latter arises in particular in two situations: One might want to hide the exact definition of a type member from client code, to obtain a kind of encapsulation known from SML-style module systems. Or one might want to override the type covariantly in subclasses to obtain family polymorphism.

Could one also go the other way, encoding abstract types with generics? It turns out that this is much harder, and that it requires at least a global rewriting of the program. This was shown by studies in the domain of module systems where both kinds of abstraction are also available [20]. Furthermore in a system with bounded polymorphism, this rewriting might entail a quadratic expansion of type bounds [4]. In fact, these difficulties are not surprising if one considers the type-theoretic foundations of both systems. Generics (without F-bounds) are expressible in System $F_{<:}$ [7] whereas abstract types require systems based on dependent types. The latter are generally more expressive than the former; for instance νObj with its path-dependent types can encode $F_{<:}$.