Type Classes as Objects and Implicits

Bruno C. d. S. Oliveira
ROSAEC Center, Seoul National University
bruno@ropas.snu.ac.kr

Adriaan Moors Martin Odersky
EPFL
{adriaan.moors, martin.odersky}@epfl.ch

Abstract

Type classes were originally developed in Haskell as a disciplined alternative to ad-hoc polymorphism. Type classes have been shown to provide a type-safe solution to important challenges in software engineering and programming languages such as, for example, *retroactive extension* of programs. They are also recognized as a good mechanism for *concept*-based *generic programming* and, more recently, have evolved into a mechanism for type-level computation.

This paper presents a lightweight approach to type classes in object-oriented (OO) languages with generics using the CONCEPT pattern and *implicits* (a type-directed implicit parameter passing mechanism). This paper also shows how Scala's type system conspires with implicits to enable, and even surpass, many common extensions of the Haskell type class system, making Scala ideally suited for generic programming in the large.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Functional Languages, Object-Oriented Languages

General Terms Languages

Keywords Type classes, C++ concepts, Abstract datatypes, Scala

1. Introduction

Type classes were introduced in Haskell (Peyton Jones 2003) as a disciplined way of defining ad-hoc polymorphic abstractions (Wadler and Blott 1989). There are several language mechanisms that are inspired by type classes: *Isabelle's type classes* (Haftmann and Wenzel 2006), *Coq's type classes* (Sozeau and Oury 2008), *C++0X concepts* (Gregor et al. 2006), *BitC's type classes* (Shapiro et al. 2008), or *JavaGI generalized interfaces* (Wehr 2009).

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

Reprinted from OOPSLA/SPLASH'10,, [Unknown Proceedings], October 17–21, 2010, Reno/Tahoe, Nevada, USA., pp. 1–20.

Lämmel and Ostermann (2006) show that type classes are useful to solve several fundamental challenges in software engineering and programming languages. In particular type classes support *retroactive extension*: the ability to extend existing software modules with new functionality without needing to touch or re-compile the original source. Type classes are also recognized (Bernardy et al. 2008; Garcia et al. 2007; Siek and Lumsdaine 2008) as a good mechanism for *concept*-based C++ style *generic programming* (Musser and Stepanov 1988), and have more recently evolved into a mechanism for type-level computation (Chakravarty et al. 2005b; Jones 2000; Schrijvers et al. 2008).

Existing proposals for type-class-like mechanisms in OO languages are rather heavyweight. The JavaGI proposal is to extend Java with *generalized interfaces* and *generalized interface implementations*. Similarly, the C++0X concepts proposal is to extend C++ with *concepts* and *model* (or *concept-map*) declarations to express concept-interfaces and their implementations. In some sense the additional constructs in JavaGI and C++0X overlap with conventional OO interfaces and classes, which play similar roles for defining the interfaces of objects and their implementations.

Type classes comprise various language constructs that can be understood in isolation. The first role of type classes is to define concepts: a set of requirements for the type parameters used by generic algorithms. For example, a sorting algorithm on lists can be expressed generically, for any elements of type T, provided that we know how to compare values of type T. One way to achieve this in an OO language with generics is to define the sorting function as follows:

$$\textbf{def}\ sort[T]\ (xs : List[T])\ (ordT : Ord[T]) : List[T]$$

In this definition, the role of *ordT* is to provide the comparison function for elements of type T. The $Ord[T]$ interface, expressed as a trait in Scala,

```
trait Ord[T] {
  def compare (a:T, b:T):Boolean
}
```

defines the ordering concept. Concepts are implemented for particular types by a *model*, which corresponds to a type class instance in Haskell, or an object in Scala.

```
object intOrd extends Ord[Int] {
  def compare (a: Int, b: Int): Boolean = a ⩽ b
}
```

However, this simple OO approach has one important limitation in practice: constraints such as *ordT* have to be explicitly passed to generic algorithms, like any other arguments. While for the definition of *sort* above this may not look too bad, many generic algorithms require multiple constraints on their type parameters, and passing all of these explicitly is cumbersome.

The second role of type classes is to propagate constraints like *ordT* automatically, making generic algorithms convenient and practical to use. Scala took inspiration from type classes and introduced *implicits*: a mechanism for implicitly passing arguments based on their types. Thus, in Scala, the ordering constraint can be implicitly passed by adding an **implicit** qualifier before the argument:

```
def sort[T] (xs: List[T]) (implicit ordT: Ord[T]): List[T]
```

Likewise potential candidate models can be considered by the compiler by being qualified with an **implicit** keyword:

```
implicit object intOrd extends Ord[Int] ...
```

This allows a convenient use of *sort*

```
scala> sort (List (3,2,1))
List (1,2,3)
```

just like the Haskell program using type classes. Furthermore, *sort* can be called with an additional ordering argument such as: *sort* (*List* (3,2,1)) (*mySpecialOrd*), where *mySpecialOrd* is another model of the ordering concept for integers. This is useful for resolving ambiguities: it is reasonable to have various orderings for the same type.

In a way type-class-style concepts provide a service like *F-bounded polymorphism* (Canning et al. 1989), which is found in conventional OO languages like Java, C# or Scala. Unlike type-parameter bounds, which impose constraints directly on the values of the bounded type, concepts like *Ord*[*T*] provide the evidence that *T* satisfies the constraints *externally*. The drawback of concept-style constraints is that dynamic dispatching over the instances of *T* is not available, but in return support for *multi-type* concepts is better and *retroactive modeling* of concepts becomes possible.

Contributions We describe a lightweight approach to type classes that can be employed in any object-oriented language that supports generics. We capture the essence of type class programming as the CONCEPT pattern, and show how implicits make the pattern practical to use. We illustrate our approach using several applications and offer an answer to Cook (2009)'s dinner quiz on the relation between objects and ADTs: in an OO language with generics, ADT signatures can be viewed as a particular class of objects.

We should note that implicits have been part of Scala for a while now (Moors et al. 2008; Odersky et al. 2006) and, in the Scala community, the encoding of type classes using implicits is folklore. However, as so often with folklore, it was never written down coherently, while the more advanced features have not been documented at all: later sections of this paper describe Scala's answer to overlapping instances (Peyton Jones et al. 1997), associated types (Chakravarty et al. 2005b), as well as how Scala's approach to type classes has surpasses Haskell type classes in some ways. These advanced features are used to introduce the idea of specifying relations on types using implicits, which is illustrated through several examples. Finally, we show that Scala has excellent support for generic programming.

Running the examples Most examples compile as-is using Scala 2.8.0. Some of the more advanced ones rely on experimental support for dependent method types, which must be enabled using the *−Xexperimental* switch. Unfortunately, some examples are affected by bugs related to the interaction between dependent method types and implicits. These are fixed in a development branch[1], which will be merged into trunk shortly, and thus appear in nightly builds leading up to the 2.8.1 release.

2. Type Classes in Haskell

This section introduces Haskell type classes as originally proposed by Wadler and Blott (1989), as well as some simple, common extensions.

2.1 Single-parameter type classes

The original model of type classes consists of single parameter type classes, which enables the definition of ad-hoc overloaded functions like *comparison*, *pretty printing* or *parsing*.

```
class Ord a where
  (⩽) :: a → a → Bool
class Show a where
  show :: a → String
class Read a where
  read :: String → a
```

A type class declaration consists of: a class name such as *Ord*, *Show* or *Read*; a type parameter; and a set of method declarations. Each of the methods in the type class declaration should have at least one occurrence of the type parameter in their signature (either as an argument or as a return type). If we think of the type parameter *a* in these type class declarations as the equivalent of the *self* argument in an OO language, we can see that a few different types of methods can be modeled:

[1] `http://github.com/adriaanm/scala/tree/topic/retire_debruijn_depmet`

- *Consumer methods* like *show* are the closest to typical OO methods. They take one argument of type a, and using that argument they produce some result.
- *Binary methods* like $\leqslant$ can take two arguments of type a, and produce some result. This appears to show some contrast with OO programming, since it is well-known that binary (and n-ary methods in general) are hard to deal with (Bruce et al. 1995). However, we should note that type class binary method arguments are only *statically dispatched* and not *dynamically dispatched*.
- *Factory methods* such as *read* return a value of type a instead of consuming values of that type. In OOP factory methods can be dealt with in different ways (for example, by using *static methods*).

Type class declarations express generic programming concepts (Bernardy et al. 2008), and the models (or implementations) of these concepts are given by type classes instances. For example

$$\textbf{instance}\ (\mathit{Ord}\ a, \mathit{Ord}\ b) \Rightarrow \mathit{Ord}\ (a,b)\ \textbf{where}$$
$$(xa,xb) \leqslant (ya,yb) = xa < ya \vee (xa \equiv ya \wedge xb \leqslant yb)$$

declares a model of the ordering concept for pairs. In this case, the ordering model itself is parametrized by an ordering model for each of the elements of the pair. With the ordering constraints we can define a generic sorting function:

$$\mathit{sort} :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a]$$

This means *sort* takes a list of elements of an arbitrary type a and returns a list of the same type, as long as the type of the elements is in the *Ord* type class, hence the *Ord* $a \Rightarrow$ context. A call to *sort* will only type check if a suitable type class instance can be found. Other than that, the caller does not need to worry about the type class context, as shown in the following interaction with a Haskell interpreter:

$$\mathit{Prelude} > \mathit{sort}\ [(3,5),(2,4),(3,4)]$$
$$[(2,4),(3,4),(3,5)]$$

One instance per type A characteristic of (Haskell) type classes is that only one instance is allowed for a given type. For example, the alternative ordering model for pairs

$$\textbf{instance}\ (\mathit{Ord}\ a, \mathit{Ord}\ b) \Rightarrow \mathit{Ord}\ (a,b)\ \textbf{where}$$
$$(xa,xb) \leqslant (ya,yb) = xa \leqslant ya \wedge xb \leqslant yb$$

in the same program as the previous instance is forbidden because the compiler automatically picks the right type class instance based on the type parameter of the type class. Since in this case there are two type class instances for the same type, there is no sensible way for the compiler to choose one of these two instances.

2.2 Common extensions

Multiple-parameter type classes A simple extension to Wadler and Blott's proposal are multiple parameter type-classes (Peyton Jones et al. 1997), which lifts the restriction of a single type parameter:

$$\textbf{class}\ \mathit{Coerce}\ a\ b\ \textbf{where}$$
$$\mathit{coerce} :: a \rightarrow b$$
$$\textbf{instance}\ \mathit{Coerce\ Char\ Int}\ \textbf{where}$$
$$\mathit{coerce} = \mathit{ord}$$
$$\textbf{instance}\ \mathit{Coerce\ Float\ Int}\ \textbf{where}$$
$$\mathit{coerce} = \mathit{floor}$$

The class *Coerce* has two type parameters a and b and defines a method *coerce*, which converts a value of type a into a value of type b. For example, by defining instances of this class, we can define coercions from characters to integers and from floating point numbers to integers.

Overlapping instances Another common extension of type classes allows instances to overlap (Peyton Jones et al. 1997), as long as there is a most specific one. For example:

$$\textbf{instance}\ \mathit{Ord}\ a \Rightarrow \mathit{Ord}\ [a]\ \textbf{where}\ \ldots$$
$$\textbf{instance}\ \mathit{Ord}\ [\mathit{Int}]\ \textbf{where}\ \ldots$$

Despite two possible matches for $[\mathit{Int}]$, the compiler is able to make an unambiguous decision to which of these instances to pick by selecting the *most specific one*. In this case, the second instance would be selected.

3. Implicits

This section introduces the Scala implementation of implicits and shows how implicits provide the missing link for type class programming to be convenient in OO languages with generics.

3.1 Implicits in Scala

Scala automates type-driven selection of values with the **implicit** keyword. A method call may omit the final argument list if the method definition annotated that list with the **implicit** keyword, and if, for each argument in that list, there is exactly one value of the right type in the *implicit scope*, which roughly means that it must be accessible without a prefix. We will describe this in more detail later.

To illustrate this, the following example introduces an **implicit** value *out*, and a method that takes an implicit argument $o : \mathit{PrintStream}$. The first invocation omits this argument, and the compiler will infer *out*. Of course, the programmer is free to provide an explicit argument, as illustrated in the last line.

$$\textbf{import}\ \mathit{java.io.PrintStream}$$
$$\textbf{implicit val}\ \mathit{out} = \mathit{System.out}$$
$$\textbf{def}\ \mathit{log}\ (\mathit{msg} : \mathit{String})\ (\textbf{implicit}\ o : \mathit{PrintStream})$$
$$= o.\mathit{println}\ (\mathit{msg})$$
$$\mathit{log}\ (\texttt{"Does not compute!"})]$$
$$\mathit{log}\ (\texttt{"Does not compute!!"})\ (\mathit{System.err})$$

Note that the arguments in an implicit argument list are part of the implicit scope, so that implicit arguments are propagated naturally. In the following example, *logTm*'s implicit argument *o* is propagated to the call to *log*.

```
def logTm (msg: String) (implicit o: PrintStream): Unit
  = log ("[" + new java.util.Date () + "]" + msg)
```

The implicit argument list must be the last argument list and it may either be omitted or supplied in its entirety. However, there is a simple idiom to encode a wildcard for an implicit argument. To illustrate this with our running example, suppose we want to generalize *logTm* so that we can specify an arbitrary prefix, and that we want that argument to be picked up from the implicit scope as well.

```
def logPrefix (msg: String)
  (implicit o: PrintStream, prefix: String): Unit
  = log ("[" + prefix + "]" + msg)
```

Now, with the following definition of the polymorphic method ?,

```
def ?[T] (implicit w: T): T = w
```

which "looks up" an implicit value of type T in the implicit scope, we can write *logPrefix* ("a") (?, "pre"), omitting the value for the output stream, while providing an explicit value for the prefix. Type inference and implicit search will turn the call ? into ?[*PrintStream*] (*out*), assuming *out* is in the implicit scope as before.

Implicit scope When looking for an implicit value of type T, the compiler will consider implicit value definitions (definitions introduced by **implicit val**, **implicit object**, or **implicit def**), as well as implicit arguments that have type T and that are in scope locally (accessible without prefix) where the implicit value is required. Additionally, it will consider implicit values of type T that are defined in the types that are part of the type T, as well as in the companion objects of the base classes of these parts. The set of parts of a type T is determined as follows:

- for a compound type T_1 **with** ... **with** T_n, the union of the parts of T_i, and T,
- for a parameterized type $S[T_1, \ldots, T_n]$, the union of the parts of S and the parts of T_i,
- for a singleton type p.**type**, the parts of the type of p,
- for a type projection $S\#U$, the parts of S as well as $S\#U$ itself,
- in all other cases, just T itself.

Figure 1 illustrates the local scoping of implicits. Two models for *Monoid*[*Int*] exist, but the scope each of them is limited to the enclosing declaration. Thus the definition of *sum* in object *A* will use the *sumMonoid* implicit. Similarly, in the object *B*, *product* will use the *prodMonoid* implicit.

```
trait Monoid[A] {
  def binary_op (x: A, y: A): A
  def identity              : A
}
def acc[A] (l: List[A]) (implicit m: Monoid[A]): A =
  l.foldLeft (m.identity) ((x,y) ⇒ m.binary_op (x,y))
object A {
  implicit object sumMonoid extends Monoid[Int] {
    def binary_op (x: Int, y: Int) = x + y
    def identity                   = 0
  }
  def sum (l: List[Int]): Int = acc (l)
}
object B {
  implicit object prodMonoid extends Monoid[Int] {
    def binary_op (x: Int, y: Int) = x * y
    def identity                   = 1
  }
  def product (l: List[Int]): Int = acc (l)
}
val test: (Int, Int, Int) = {
  import A._
  import B._
  val l = List (1,2,3,4,5)
  (sum (l), product (l), acc (l) (prodMonoid))
}
```

Figure 1. Locally scoped implicits in Scala.

Furthermore, both implicits can be brought into scope using **import** and the user can choose which implicit to use by explicitly parameterizing a declaration that requires a monoid. Thus, the result of executing *test* is $(15, 120, 120)$. Note that, if instead of *acc* (l) (*prodMonoid*) we had used *acc* (l) in the definition of *test*, an ambiguity error would be reported, because two different implicit values of the same type (*prodMonoid* and *sumMonoid*) would be in scope.

Implicit search and overloading To determine an implicit value, the compiler searches the implicit scope for the value with the required type. If no implicit can be found for an implicit argument with a default value, the default value is used. If more than one implicit value has the right type, there must be a single "most specific" one according to the following ordering, which is defined in more detail by (Odersky 2010, 6.26.3).

An alternative A is *more specific* than an alternative B if the relative weight of A over B is greater than the relative weight of B over A. The relative weight is a score between 0 and 2, where A gets a point over B for being as specific as B,

and another if it is defined in a class (or in its companion object) which is derived from the class that defines B, or whose companion object defines B. Roughly, a method is as specific as a member that is applicable to the same arguments, a polymorphic method is compared to another member after stripping its type parameters, and a non-method member is as specific as a method that takes arguments or type parameters.

Finally, termination of implicit search is ensured by keeping track of an approximation of the types for which an implicit value has been searched already (Odersky 2010, 7.2).

3.2 Implicits as the missing link

Implicits provide the type-driven selection mechanism that was missing for type class programming to be convenient in OO. For example, the *Ord* type class and the pair instance that was presented in Section 2 would correspond to:

```
trait Ord[T] {
  def compare (x:T, y:T) : Boolean
}
implicit def OrdPair[A,B]
  (implicit ordA: Ord[A], ordB: Ord[B])
    = new Ord[(A,B)] {
      def compare (xs: (A,B), ys: (A,B)) = ...
      }
```

Note that the syntactic overhead compared to Haskell (highlighted in gray) includes useful information: type class instances are named, so that the programmer may supply them explicitly to resolve ambiguities manually.

The *cmp* function is rendered as the following method:

```
def cmp[a] (x:a, y:a) (implicit ord: Ord[a]) : Boolean
  = ord.compare (x,y)
```

This common type of implicit argument can be abbreviated using context bounds (highlighted in gray):

```
def cmp[a:Ord] (x:a, y:a) : Boolean
  = ?[Ord[a]].compare (x,y)
```

Since we do not have a name for the type class instance anymore, we use the ? method to retrieve the implicit value for the type $Ord\,[a]$. Note that the same shorthand can be used in the *OrdPair* method above, reducing the syntactic noise. Furthermore, the *cmp* provides a slightly more terse interface to the *Ord* type class in the sense that a client can now call $cmp\ ((3,4),(5,6))$ instead of $?[Ord\,[(Int,Int)]].compare\ ((3,4),(5,6))$.

The pimp-my-library pattern Neither $cmp\ ((3,4),(5,6))$ or $?[Ord\,[(Int,Int)]].compare\ ((3,4),(5,6))$ are idiomatic in OOP. The 'pimp my library' pattern (Odersky 2006) uses implicits to allow the more natural $x.compare\ (y)$, assuming the type of x does not define the *compare* method. In Scala, implicit values that have a function type act as implicit *conversions*. For a method call such as $x.compare\ (y)$ to be well-typed, the type of x must either define a suitable *compare* method, or there must be an implicit conversion c in scope so that $(c\ (x)).compare\ (y)$ is well-typed without further use of implicit conversions. Thus, it suffices to define an implicit method (the compiler converts methods to functions when needed) *mkOrd* that will convert a value of a type that is in the *Ord* type class into an object that has the expected interface:

```
implicit def mkOrd[T:Ord] (x:T) : Ordered[T]
  = new Ordered[T] {
      def compare (o:T) = ?[Ord[T]].compare (x,o)
    }
```

Leaving off the target of the comparison in the *compare* method, which has been passed to the implicit conversion *mkOrd*, *Ordered* is defined as:

```
trait Ordered[T] {
  def compare (o:T) : Boolean
}
```

```
trait Ord[T] {
  def compare (x:T, y:T) : Boolean
}
class Apple (x:Int) {}
object ordApple extends Ord[Apple] {
  def compare (a1:Apple, a2:Apple) = a1.x ⩽ a2.x
}
def pick[T] (a1:T, a2:T) (ordA: Ord[T]) =
  if (ordA.compare (a1,a2)) a2 else a1
val a1 = new Apple (3)
val a2 = new Apple (5)
val a3 = pick (a1,a2) (ordApple)
```

Figure 2. Apples to Apples with the CONCEPT pattern.

4. The CONCEPT Pattern

This section introduces the CONCEPT pattern, which is inspired by the basic Haskell type classes presented in Section 2 and C++ concepts. This pattern can be used in any OO language that supports generics, such as current versions of Java or C#. However, without support for implicits, some applications can be cumbersome to use due to additional parameters for the constraints.

4.1 Concepts: type-class-style interfaces in OO

The CONCEPT pattern aims at representing the generic programming notion of concepts as conventional OO interfaces

with generics. Concepts describe a set of requirements for the type parameters used by generic algorithms. Figure 2 shows a small variation of the *apples-to-apples* motivational example for concepts (Garcia et al. 2007). This example serves the purpose of illustrating the different actors in the CONCEPT pattern. The trait $Ord\,[T]$ is an example of a *concept interface*. The type argument T of a concept interface is the *modeled type*; an *Apple* is a *concrete modeled type*. Actual objects implementing concept interfaces such as *ordApple* are called *models*. Finally, whenever ambiguity arises, we will refer to methods in a concept interface as *conceptual methods* to distinguish them from conventional methods defined in the modeled type.

The CONCEPT pattern can model n-ary, factory and consumer methods just like type classes. Concept interfaces for the type classes *Show* and *Read* presented in Section 2.1 are:

```
trait Show[T] {
  def show (x:T):String
}
trait Read[T] {
  def read (x:String):T
}
```

The *printf* example in Section 2.1 presents an example of a concept-interface with a factory method. Most of the examples in this paper involve consumer methods.

Multi-type Concepts Using standard generics it is possible to model multi-type concepts. That is, concepts that involve several different modeled types. For example, the *Coerce* type class in Section 2.2 can be expressed as a concept interface as:

```
trait Coerce[A,B] {
  def coerce (x:A):B
}
```

The *zipWithN* example (in Section 6.4) and generalized constraints (in Section 6.6) provide applications of multi-type concepts.

Benefits of the* CONCEPT *pattern The CONCEPT pattern offers the following advantages:

1. *Retroactive modeling:* The CONCEPT pattern allows mimicking the addition of a method to a class without having to modify the original class. For example, in Figure 2, the declaration of *Apple* did not require any knowledge about the *compare* functionality upfront. The *ordApple* model adds support for such method externally.
2. *Multiple method implementations:* It is possible to have multiple implementations of conceptual methods for the same type. For example, an alternative ordering model for apples can be provided:

```
object ordApple2 extends Ord[Apple] {
  def compare (a1:Apple, a2:Apple) = a1.x > a2.x
}
```

3. *Binary (or n-ary) methods:* Conceptual methods can have multiple arguments of the manipulated type. Thus a simple form of type-safe statically dispatched n-ary methods is possible.
4. *Factory methods:* Conceptual methods do not need an actual instance of the modeled type. Thus they can be factory methods.

Limitations and Alternatives The main limitation of the CONCEPT pattern is that all arguments of conceptual methods are *statically dispatched*. Thus, conceptual methods are less expressive than conventional OO methods, which allow the self-argument to be dynamically dispatched, or multi-methods (Chambers and Leavens 1995), in which all arguments are dynamically dispatched.

Bounded polymorphism offers an alternative to type-class-style concepts. With bounded polymorphism the apples-to-apples example could be modeled as follows:

```
trait Ord[T] {
  def compare (x:T):Boolean
}
class Apple (x:Int) extends Ord[Apple] ...
```

The main advantage of this approach is that *compare* is a real, dynamically dispatched, method of *Apple*, and all the private information about *Apple* objects is available for the method definition. However, with this alternative, modeled types such as *Apple* have to state upfront which concept interfaces they support. This precludes retroactive modeling and makes it harder to support multiple implementations of a method for the same object. Multi-type concepts are possible, but they can be quite cumbersome to express and they can lead to a combinatorial explosion on the number of concept interfaces (Järvi et al. 2003). Factory methods can be supported with this approach through a static method, although this dictates a single implementation. Binary methods such as *compare* are also possible, although they are asymmetric in the sense that the first argument is dynamically dispatched, whereas the second argument is statically dispatched.

Language Support In languages such as Java or C# concepts need to be explicitly passed as in Figure 2. In Scala it is possible to pass concepts implicitly as shown in Section 3.2. Additionally, in languages like Java or C#, there is some syntactic noise because the method cannot be invoked directly on the manipulated object:

```
a = new Apple (3);
a.compare (new Apple (5));
```

is invalid. Instead, we must write:

```
a = new Apple (3);
ordApple.compare (a, new Apple (5));
```

In Scala, as discussed in Section 3.2, it is possible to eliminate this overhead using implicits. All that is needed is 1) mark the models (and any possible constraints) with **implicit** and 2) create a simple interface for the comparison function that takes the ordering object implicitly. Thus provided that the apples-to-apples is modified as follows:

```
implicit object ordApple extends Ord[Apple] ...
def cmp[A] (x:A, y:A) (implicit ord: Ord[A]) =
  ord.compare (x, y)
```

Then we can write:

```
a = new Apple (3);
cmp (a, new Apple (5));
```

Modifying *pick* similarly,

```
def pick[T:Ord] (a1:T, a2:T) =
  if (cmp (a1, a2)) a2 else a1
```

allows rewriting the value a_3 as:

```
val a3 = pick (a1, a2)
```

In C#, *extension methods* provide language support for (statically dispatched) retroactive implementations. Haskell type classes, JavaGI and the C++0X concepts proposal provide direct language support for concept-style interfaces. JavaGI's *generalized interfaces* offer more expressiveness than the CONCEPT pattern. JavaGI's retroactive implementations support multi-methods on the instances of the manipulated types.

The Scala approach to concept-style interfaces is to express them with the CONCEPT pattern and implicits. This makes the pattern very natural to use without an additional, pattern-specific, language construct.

5. Applications and Comparison with Type Classes

The CONCEPT pattern has several applications, including some that go beyond concepts' traditional application, constrained polymorphism. The pattern is illustrated by example in the next few subsections. This section also compares the programs written with the CONCEPT pattern with similar programs using type classes. To help in this comparison, the significant differences between the OO programs and the equivalent programs using type classes are marked in gray. The corresponding Haskell code is available in Appendix A.

```
trait Eq[T] {
  def equal (a:T, b:T) : Boolean
}
trait Ord[T] extends Eq[T] {
  def compare (a:T, b:T) : Boolean
  def equal (a:T, b:T) : Boolean =
    compare (a, b) ∧ compare (b, a)
}
class IntOrd extends Ord[Int] {
  def compare (a:Int, b:Int) = a ⩽ b
}
class ListOrd[T] (ordD: Ord[T]) extends Ord[List[T]] {
  def compare (l1:List[T], l2:List[T]) =
    (l1, l2) match {
      case (x::xs, y::ys) ⇒
        if (ordD.equal (x, y)) compare (xs, ys)
        else ordD.compare (x, y)
      case (_, Nil)       ⇒ false
      case (Nil, _)       ⇒ true
    }
}
class ListOrd2[T] (ordD: Ord[T]) {
  extends Ord[List[T]] {
  private val listOrd = new ListOrd[T] (ordD)
  def compare (l1:List[T], l2:List[T]) =
    (l1.length < l2.length) ∧ listOrd.compare (l1, l2)
}
```

Figure 3. Equality and ordering concepts and some models.

5.1 Ordering concept

Figure 3 shows how to implement an ordering concept using the CONCEPT pattern. This concept is similar to the one used in Figure 2, except that it introduces an equality concept $Eq[T]$. For convenience, we use subtyping to express the refinement relation between the concepts *Eq* and *Ord*, although delegation, which is closer to the typical Haskell implementation (see Figure 7), would work as well. In conventional OO languages such as Java or C# the traits *Eq* and *Ord* would correspond to interfaces. Thus, the default definition for equality in the *Ord* trait would not be definable directly on the interface. Scala traits offer a very convenient way to express such default definitions, but such functionality can be mimicked in other ways in Java or C#. The classes *IntOrd*, *ListOrd* and *ListOrd2* define three models of *Ord*; the first one for integers and the other two for lists.

```
def cmp[T] (x:T, y:T) (implicit ord:Ord[T]) =
  ord.compare (x, y)
implicit val IntOrd = new Ord[Int] {...}
implicit def ListOrd[T] (implicit ordD:Ord[T]) =
  new Ord[List[T]] {...}
def ListOrd2[T] (implicit ordD:Ord[T]) =
  new Ord[List[T]] {...}
```

Figure 4. Variation of the Ordering solution using implicits.

The three models illustrate the retroactive capabilities of the CONCEPT pattern: the models are added after *Int* and *List*[*T*] have been defined. The two models for lists illustrate that multiple models can co-exist at the same time.

Comparison with Type Classes The essential difference between the OO code in Figure 3 and the similar definitions using type classes (which can be found in Figure 13) is that models, and model arguments, need to be named. In Haskell, instances can be viewed as a kind of *anonymous objects*, which only the compiler gets direct access to. This partly explains why the definition of *ListOrd2* is grayed out: in Haskell two instances for the same modeled type are forbidden.

In the OO version, it is necessary to first create the models explicitly. For example:

```
def sort[T] (xs:List[T]) (ordT:Ord[T]):List[T] = ...
val l1 = List (7,2,6,4,5,9)
val l2 = List (2,3)
val test  = new ListOrd (new IntOrd ()).compare (l1,l2)
val test2 = new ListOrd2 (new IntOrd ()).compare (l1,l2)
val test3 = sort (l1) (new ListOrd (new IntOrd ()))
```

In the type class version, the equivalent code would be:

```
sort :: Ord t => [t] -> [t]
l1 = [7,2,6,4,5,9]
l2 = [2,3]
test  = compare l1 l2
test3 = sort l1
```

Clearly, in the OO version, the use of *compare* in *test* and *test*$_2$ is less convenient than simply calling *compare* l_1 l_2, but it does offer the possibility of switching the implementation of the comparison operation in *test*$_2$. In *test*$_3$ creating the models explicitly is also somewhat verbose and inconvenient.

Solution using implicits The convenience of type classes can be recovered with implicits. Figure 4 shows a variation of the code in Figure 3. Only the differences are shown: definitions are used instead of conventional OO classes to define the models for *Ord*; and we use a definition *cmp* to provide a nice interface to the *compare* method. The first two models are implicit, but *ListOrd2* cannot be made implicit because it would clash with *ListOrd*. The client code for the test functions is simplified, being comparable to the version with Haskell type classes. Furthermore, it is still possible to define *test*$_2$.

```
val test  = cmp (l1,l2)
val test2 = cmp (l1,l2) (ListOrd2)
val test3 = sort (l1)
```

```
trait Set[S] {
  val empty:S
  def insert (x:S, y:Int):S
  def contains (x:S, y:Int):Boolean
  def union (x:S, y:S):S
}
class ListSet extends Set[List[Int]] {
  val empty = List ()
  def insert (x:List[Int], y:Int) = y::x
  def contains (x:List[Int], y:Int) = x.contains (y)
  def union (x:List[Int], y:List[Int]) = x.union (y)
}
class FunctionalSet extends Set[Int => Boolean] {
  val empty = (x:Int) => false
  def insert (f:Int => Boolean, y:Int) =
    z => y.equals (z) || f (z)
  def contains (f:Int => Boolean, y:Int) = f (y)
  def union (f:Int => Boolean, g:Int => Boolean) =
    y => f (y) || g (y)
}
```

Figure 5. An ADT signature and two implementations.

5.2 Abstract data types

Cook (2009) shows that type classes can be used to implement what is effectively the algebraic signature of an Abstract Data Type (ADT). Programs using these type classes in a certain disciplined way have the same abstraction benefits as ADTs. Exploiting this observation, we now show a simple and practical encoding of ADTs in an object-oriented language with generics using the CONCEPT pattern. ADT signatures show an application of the pattern that is different from how concepts are traditionally used. Additionally, it illustrates why passing a model explicitly is sometimes desirable.

Figure 5 models an ADT signature for sets of integers using the CONCEPT pattern. The trait *Set*[*S*], the concept interface, defines the ADT signature for sets. The type *S* is

the modeled type. The method *empty* is an example of a factory method: a new set is created without any previous set instance. The methods *insert* and *contains* are examples of consumer methods: they act on existing instances of sets to achieve their goal. Finally *union* provides an example of a binary method: two set instances are needed to take their union. Two alternative models are shown: *ListSet*, using a lists to model sets; and *FunctionalSet*, which uses a boolean predicate instead.

The client programs using models of ADT signatures can be used in a very similar way to ADTs implemented using ML modules (MacQueen 1984). For example:

$$\textbf{val } setImpl_1 = \textbf{new } ListSet\ ()$$
$$\textbf{val } setImpl_2 = \textbf{new } FunctionalSet\ ()$$
$$\textbf{def } test_1\,[S]\ (s{:}Set\,[S]) : Boolean = \\ s.contains\ (s.insert\ (s.insert\ (s.empty, 5), 6), 6)$$

In this case two different implementations of sets, $setImpl_1$ and $setImpl_2$, are created. The definition $test_1$ takes a set implementation and defines a program using that implementation. Importantly, the set implementation is *polymorphic* on S, which means that *any* subclass of $Set\,[S]$ will be valid as an argument. In particular, both $test_1\ (setImpl_1)$ and $test_1\ (setImpl_2)$ are valid arguments.

A reasonable question to ask at this point is whether the programs written with ADT signatures are actually related to conventional programs using ADTs. We discuss this issue next.

Where is the existential? In their seminal paper on ADTs, Mitchell and Plotkin (1988) show that "abstract types have existential type". Formally an ADT can be viewed has two distinct parts: the *ADT signature* and the *existential* encapsulating the type of the concrete representation:

$$SetADT = \exists\, S.\ Set\,[S]$$

The trait $Set\,[S]$ defines only the signature, but the existential, which provides information hiding, is missing. This means that certain programs can exploit the concrete representation, breaking encapsulation. Still, as Cook observes, it is possible to enforce information hiding with some discipline and the help of the type system; if client programs do not exploit the concrete representations of S then the same benefits of ADTs apply. To see why this is the case consider the equivalent type-theoretic version of the $test_1$ program:

$$test_1 : SetADT \to Boolean$$
$$test_1 = \lambda s \to \\ s.contains\ (s.insert\ (s.insert\ (s.empty, 5), 6), 6)$$

Unfolding the *SetADT* type into its definition yields

$$test_1 : (\exists\, S.\ Set\,[S]) \to Boolean$$

and this type is isomorphic to

$$test_1 : \forall\, S.\ Set\,[S] \to Boolean$$

which is the type-theoretic type corresponding to the Scala type of $test_1$.

In other words, $test_1$ has the existential type that provides the information hiding of the equivalent program with ADTs. While it is certainly debatable whether or not the existential should be placed in the actual ADT definition, concept interfaces are a simple way to encode ADT-like programs in any OO language with generics. This provides an alternative answer to Cook's dinner quiz on the relationship between objects an ADTs: in an OO language with generics, ADT signatures can be viewed as concept interfaces, and implementations of these signatures can be modeled as objects.

Comparison with Type Classes There are no significant differences between the OO version of the program and the Haskell version (which can be found in Figure 14), except that the models need to be named. However, client code is more interesting to compare. While in the OO version we write:

$$\textbf{def } test_1\,[S]\ (s{:}Set\,[S]) : Boolean = \\ s.contains\ (s.insert\ (s.insert\ (s.empty, 5), 6), 6)$$

the Haskell version of this code

$$test_1 :: Set\ s \Rightarrow Bool$$
$$test_1 = contains\ (insert\ (insert\ empty\ 5)\ 6)$$

does not work. The problem is that this program is *ambiguous*: since Haskell type classes work under the assumption that every dictionary is inferred by the compiler, there is no straightforward way to tell $test_1$ which specific instance of *Set* is to be used.

Programs using ADT-like structures show how certain programs do not fit well with the implicit nature of type classes. To be more precise, ADTs fit very well within the "class" of programs that type classes capture, however explicitly passing "type class instances" (the models of ADT signatures) is desirable. The CONCEPT pattern solution is better in this respect because the models can be explicitly passed.

5.3 Statically-typed printf

Our final application of the CONCEPT pattern is a statically-typed version of the C *printf* function similar to the one presented by (Kennedy and Russo 2005). This example shows that often it is possible to model structures resembling extensible (in the sense that new constructors can be added) *generalized algebraic datatypes* (GADTs) (Peyton Jones et al. 2006) using the CONCEPT pattern.

Figure 6 shows the implementation of a simple version of the C-style *printf* function using the CONCEPT pattern. The implementation exploits an insight by (Danvy 1998), who realized that by changing the representation of the format string, it is possible to encode *printf* in a conventional

```
trait Format[A] {
  def format (s: String): A
}
def printf[A] (format: Format[A]): A =
  format.format ("")
class I[A] (formatD: Format[A])
  extends Format[Int ⇒ A] {
  def format (s: String) = i ⇒
    formatD.format (s + i.toString)
}
class C[A] (formatD: Format[A])
  extends Format[Char ⇒ A] {
  def format (s: String) = c ⇒
    formatD.format (s + c.toString)
}
class E extends Format[String] {
  def format (s: String) = s
}
class S[A] (l: String, formatD: Format[A])
  extends Format[A] {
    def format (s: String) = formatD.format (s + l)
}
```

Figure 6. Printf as an instance of the CONCEPT pattern

Hindley-Milner type system. The basic idea of the OO version is to use a concept interface *Format* to represent the format string of *printf*. Noteworthy, the *format* conceptual method is a factory: it creates instances of the modeled types. Four different models are provided: one for integers, another for characters, a termination string and string literals. One advantage of this solution is that it is easy to introduce new format specifiers simply by creating a new model of the *Format* concept.

Comparison with Type Classes Like with the previous two examples, the models and model arguments need to be named in the OO version. An important difference is that, with type classes, we cannot implement a corresponding instance for the $S[A]$ model. The problem is that, to define *S*, a *String* argument is required but type class instances can only take type class dictionaries in the instance constraints. Thus, the following is not allowed:

```
instance (String, Format a) ⇒ Format a where ...
```

Another difference concerns the client code. In the OO version, the formatting string needs to be explicitly constructed and passed. This has both advantages and disadvantages. The advantage is that the format string can be chosen precisely, as it it the case for the standard *printf* function.

```
val fmt: Format[Int ⇒ Char ⇒ String] =
  new S ("Int: ", new I (new S (" Char: ",
    new C (new S (".", new E)))))
val test = printf (fmt) (3) ('c')
```

For example, we can construct format strings using the instances of the *S* class. In Haskell, such flexibility is not easily available. Nevertheless, if such flexibility is not required, the dictionary is inferred in Haskell, making the similar programs more compact.

```
test :: String
test = printf (3 :: Int) 'c'
```

Finally, we should note that if we modify the OO version into a more idiomatic version using Scala's implicits (as done for the ordering example in Section 5.1), then we can also infer the same format strings as in the Haskell version.

5.4 Type class programs are OO programs

As we have seen so far, programs written with type classes seem to have a close correspondence with OO programs. Still, how can we more precisely pinpoint the relationship between a Haskell type class program and an OO program?

One answer to this question, which we describe next, lies on the relationship between the *dictionary translation* (Wadler and Blott 1989) of Haskell type classes and a simple form of the functional *recursive records* (Cook and Palsberg 1994) encoding of OO programs.

The dictionary translation, which is used by most Haskell compilers, converts a program using type classes into a program using no type classes. This translation is necessary because type classes are a language mechanism of the *source language*, but most Haskell compilers use core languages (usually a variant of System F), which does not have a native notion of type classes.

Figure 7 shows how the program in Figure 13 looks like after the dictionary translation. Like with the OO programs, the parts that do not have direct correspondents in the Haskell type class code are highlighted in gray. Essentially what happens is that the type class *Ord* is translated into a program using a record. Each instance becomes a value (or, more precisely, a function that takes the dictionaries for the class contexts, if any) that represents the corresponding dictionary.

In the translated code, many of the characteristics of the OO version are present. One similarity is that the "instances" and the arguments need to be named. For example, the corresponding value for the equality dictionary for lists is defined as:

```
listOrd :: Ord a → Ord [a]
listOrd ordD = Ord ...
```

```
data Ord a = Ord {
  eq      :: a → a → Bool,
  compare :: a → a → Bool
}
intOrd :: Ord Int
intOrd = Ord {
  eq      = λa b → compare intOrd a b ∧
                   compare intOrd b a,
  compare = λx y → x ⩽ y
}
listOrd :: Ord a → Ord [a]
listOrd ordD = Ord {
  eq      = λa b → compare (listOrd ordD) a b ∧
                   compare (listOrd ordD) b a,
  compare = λl1 l2 → case (l1, l2) of
    (x:xs, y:ys) →
      if (eq ordD x y)
        then compare (listOrd ordD) xs ys
        else compare ordD x y
    (_, [])      → False
    (_, _)       → True
}
```

Figure 7. Ordering after the dictionary translation.

Here, *listOrd* is the name of the dictionary constructor and *ordD* is the name of the argument of the constructor. Another similarity is that in the invocations of the *compare* methods: the dictionary value for the method is also explicit. For example,

```
compare ordD x y
```

As it turns out, the dictionary translation version of the Haskell program has so much in common with the OO version because it corresponds to a simple form of the *recursive records* functional encoding of the OO program.

The only significant difference between the dictionary translation version and the OO version, highlighted in gray next, is that the Haskell version has some explicit recursive calls on *listOrd ordD*:

```
listOrd :: Ord a → Ord [a]
listOrd ordD = Ord {
  ...
  compare = λl1 l2 → case (l1, l2) of
    (x:xs, y:ys) →
      if (eq ordD x y)
        then compare (listOrd ordD) xs ys
        else compare ordD x y
  ...
```

The recursive records interpretation of OO programs also helps explaining this (superficial) difference. In the OO program there is an implicitly passed self-argument in *compare* (xs, ys) (this is sugar for **this**.*compare* (xs, ys)) and this self-argument is a *recursive call* in the corresponding recursive records interpretation. What is happening in the dictionary translation is that recursive calls are directly used.

In summary, the relationship between type classes and OO programs is this: every type class program translated using the dictionary translation corresponds to a OO program encoded using a simple form of the recursive records functional OO encoding.

6. Advanced Uses of Type Classes

This section briefly explains GHC Haskell's associated types and shows in detail how they can be encoded in Scala using type members and dependent method types, which are also first introduced. The encoding of associated types and other advanced features of Scala (such as prioratized overlapping implicits) are illustrated with three examples: type-safe session types, an n-ary version of the *zipWith* function, and an encoding of generalized constraints. We conclude this section with a description of the pattern that is common to all of these examples.

6.1 Associated types in GHC Haskell

The Glasgow Haskell Compiler (GHC) (Peyton Jones et al. 2009) is a modern Haskell implementation that provides a type class mechanism that goes well beyond Wadler's original proposal. Of particular interest is the use of type classes for *type-level computation* using extensions such as associated types (Chakravarty et al. 2005b).

Associated types are type declarations that are associated to a type class and that are made concrete in the class's instances, just like type class methods. For example, the class *Collects* represents an interface for collections of type *c* with an associated type *Elem c*, which denotes the type of the elements of *c*.

```
class Collects c where
  type Elem c
  empty :: c
  insert :: c → Elem c → c
  toList :: c → [Elem c]
```

Two possible instances are:

```
instance Collects BitSet where
    type Elem BitSet = Char
    ...
instance (Collects c, Hashable (Elem c)) ⇒
  Collects (Array Int c) where
```

type *Elem* (*Array Int c*) = *Elem c*
...

The basic idea is that, for a *BitSet* collection, the associated element type should be characters. For arrays of values of some type *c*, the type of the elements should be the same as the type of the elements of *c* itself.

Associated types require type-level computation. In the *Collects* example this manifests itself whenever a value of type *Elem c* is needed. For example, when using *insert* the second argument has the type *Elem c*. For an array of bit sets *Array Int BitSet* the type *Elem* (*Array Int BitSet*) should be *evaluated* to *Char*, in order for the type-checker to validate suitable values for that call. This entails unfolding the associated type definitions to conclude that the element type of an array of bit sets is indeed a character.

6.2 Implicits and type members

Associated types fell out for free in Scala, with the introduction of implicits, due to existing support for type members. Before illustrating this, we briefly introduce type members, path-dependent types and dependent method types.

A type member is a type that can be selected on an object. Like its value-level counterpart, a type member may be abstract, in which case it is similar to a type parameter, or it may be concrete, serving as a type alias. For type safety, an abstract type member may only be selected on a *path* (Odersky et al. 2003), a *stable* (immutable) value.

Technically, types may only be selected on types, but a path p is readily turned into a singleton type $p.\mathbf{type}$, which is the type that is inhabited by exactly one value: the object referenced by p. The type selection $p.T$, where T is a type member defined in the type of p, is syntactic sugar for $p.\mathbf{type}\#T$. We say that a type that contains the type $p.\mathbf{type}$ *depends* on the path p. The type $p.T$ is a *path-dependent type*.

Since a method argument is considered a stable value, a type may depend on it. A method type that depends on one or more of the method's arguments is called a *dependent method type*. The simplest example of such a type arises in the following version of the *identity* method:

def *identity* $(x:\mathit{AnyRef}):x.\mathbf{type} = x$

Since values are not allowed in paths (for now), this version of *identity* must be limited to references. Using this *identity*, we can statically track that *y* and *y2* are aliases:

val $y =$ `"foo"`
val $y2:y.\mathbf{type} = \mathit{identity}\ (y)$

For now, these types must be enabled explicitly by the $-$*Xexperimental* compiler flag in Scala 2.8.

We can massage *identity* into a more precise version of the implicit argument wildcard ? that we introduced earlier:

def $?[T <: \mathit{AnyRef}]\ (\mathbf{implicit}\ w:T):w.\mathbf{type} = w$

```
def add_server =
  In {x: Int ⇒
  In {y: Int ⇒ System.out.println ("Thinking")
  Out (x + y,
  Stop ())}}
def add_client =
  Out (3,
  Out (4, {System.out.println ("Waiting")
  In {z: Int ⇒ System.out.println (z)
  Stop ()}}))
```

Figure 8. An example session.

This version of ? may be used to access the implicit value of type T. Its additional precision will be essential in the examples below, which select type members on implicit arguments. The following examples illustrate that the combination of implicit search and dependent method types is an interesting recipe for type-level combination.

6.3 Session types

As our first example, we port a type class encoding of session types (Honda 1993) by Kiselyov et al. (2009) to Scala. Session types describe the relationship between the types of a pair of communicating processes. A process is composed of a sequence of smaller processes.

For example, the server process in Figure 8 takes two integers arguments as inputs and returns the sum of these two integers as output. The corresponding client performs the dual of the server process. The example uses the elementary processes that are defined in Figure 9. The *Stop* process indicates the end of communication, whereas *In* and *Out* processes are paired to specify the flow of information during the session. An $\mathit{In}[a,b]$ process takes an input of type a, and continue with a process of type b.[2]

Since they are duals, *add_client* and *add_server* form a session. We capture the notion of duality in Figure 9. The *DualOf* relation is defined in the comments using inference rules. It is rendered in Scala as the *Session* trait, which declares the relation on its type parameter *S* (which is aliased as *Self* for convenience) and its abstract type member *Dual*, which corresponds to an associated type. Thus, an instance of type $\mathit{Session}[S]\ \{\mathbf{type}\ \mathit{Dual} = D\}$ is evidence of *S DualOf D*; such a value witnesses this fact by describing how to compose an instance of *S* with an instance of *D* (through the *run* method). *StopSession*, *InDual* and *OutDual* describe what it "means" for each of the atomic process types to be in a session with their respective duals by constructing the witness at the corresponding concrete types.

[2] The $+$ and $-$ symbols denote, respectively, co-variance and contra-variance of the type constructor in these type parameters (Emir et al. 2006).

```
sealed case class Stop
sealed case class In[−A,+B] (recv:A ⇒ B)
sealed case class Out[+A,+B] (data:A,cont:B)
trait Session[S] {type Self = S; type Dual
  type DualOf[D] = Session[Self] {type Dual = D}
  def run (self:Self,dual:Dual):Unit
}
/*
  ———————————— StopDual
  Stop DualOf Stop
*/
implicit object StopDual extends Session[Stop] {
  type Dual = Stop
  def run (self:Self,dual:Dual):Unit = {}
}
/*
        Cont DualOf ContD
  ———————————————————————————— InDual
  In[Data,Cont] DualOf Out[Data,ContD]
*/
implicit def InDual[D,C] (implicit cont:Session[C])
  = new Session[In[D,C]] {
  type Dual = Out[D,cont.Dual]
  def run (self:Self,dual:Dual):Unit =
    cont.run (self.recv (dual.data ),dual.cont)
}
/*
        Cont DualOf ContD
  ———————————————————————————— OutDual
  Out[Data,Cont] DualOf In[Data,ContD]
*/
implicit def OutDual[D,C] (implicit cont:Session[C])
  = new Session[Out[D,C]] {
  type Dual = In[D,cont.Dual]
  def run (self:Self,dual:Dual):Unit =
    cont.run (self.cont,dual.recv (self.data ))
}
```

Figure 9. Session types.

The expression *InDual* has the type[3]:

```
[D,C] (implicit cont:Session[C]) Session[In[D,C]] {
  type Dual = Out[D,cont.Dual]}
```

This is a polymorphic dependent method type, where $[D,C]$ denotes the universal quantification $\forall\, D,C.$, (**implicit** *cont*: *Session*$[C]$) describes the argument list, and

```
Session[In[D,C]] {type Dual = Out[D,cont.Dual]}
```

[3] Note that this type is not expressible directly in the surface syntax.

```
case class Zero ()
case class Succ[N] (x:N)
trait ZipWith[N,S] {
  type ZipWithType
  def manyApp:N ⇒ Stream[S] ⇒ ZipWithType
  def zipWith:N ⇒ S ⇒ ZipWithType =
    n ⇒ f ⇒ manyApp (n) (repeat (f))
}
def zWith[N,S] (n:N,s:S)
      (implicit zw:ZipWith[N,S]):zw.ZipWithType =
  zw.zipWith (n) (s)
implicit def ZeroZW[S] = new ZipWith[Zero,S] {
  type ZipWithType = Stream[S]
  def manyApp = n ⇒ xs ⇒ xs
}
implicit def SuccZW[N,S,R]
  (implicit zw:ZipWith[N,R]) =
  new ZipWith[Succ[N],S ⇒ R] {
    type ZipWithType = Stream[S] ⇒ zw.ZipWithType
    def manyApp = n ⇒ xs ⇒ ss ⇒ n match {
      case Succ (i) ⇒ zw.manyApp (i) (zapp (xs,ss))
    }
  }
```

Figure 10. N-ary *zipWith*.

is the method's (inferred) result type, which depends on its *cont* argument. Using the *DualOf* type alias, the modelled relation can be made more explicit:

```
Session[In[D,C]]#DualOf[Out[D,cont.Dual]]
```

The same type alias is used in the context bound on *runSession*'s D type parameter, which should be read as: "D must be chosen so that S is the dual of D". In our mathematical notion, this is rendered as S *DualOf* D. To run the session, the evidence of S and D being in the *DualOf* relation is recovered using the ? method.

```
def runSession[S,D:Session[S]#DualOf]
  (session:S,dual:D) =
  ?[Session[S]#DualOf[D]].run (session,dual)
def myRun = runSession (add_server,add_client)
```

6.4 Arity-polymorphic ZipWith in Scala

Typically, functional programming languages like Haskell contain implementations of *zipWith* functions for two list arguments:

```
zipWith :: (a → b → c) → [a] → [b] → [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _           = []
```

McBride (2002) generalized such definition into an n-ary *zipWith* function.

$$zipWith_n :: (a_1 \to a_2 \to \ldots \to a_n) \to ([a_1] \to [a_2] \to \ldots \to [a_n])$$

In other words, $zipWith_n$ is a function that given a function with n arguments and n lists, provides a corresponding version of the n-ary *zipWith* function. Similar challenges arise in OO libraries that model databases as collections of N-tuples, where the operations on these tuples should work for any N. A more general variant of this problem is discussed by Kiselyov et al. (2004).

This example essentially performs type-level computation, since the return type $[a_1] \to [a_2] \to \ldots \to [a_n]$ is computed from the argument type $(a_1 \to a_2 \to \ldots \to a_n)$. Figure 10 shows an implementation for the n-ary *zipWith* function in Scala. The types *Zero* and *Succ* $[N]$ are Church encodings of the natural numbers at the type level.

The *ZipWith* trait is a concept interface with two type arguments: N represents a natural number; and S is the type of the function argument of $zipWith_n$. The type member *ZipWithType* determines the return type. The *zWith* method is the interface for the n-ary zipWith function. The two definitions *ZeroZW* and *SuccZW* provide two models that, respectively, correspond to the base case (for $N = 0$) and the inductive case. The functions *repeat* and *zapp*, used in *zipWith* and *manyApp* are defined as:

```
def repeat[A] (x:A) : Stream[A] = cons (x, repeat (x))
def zapp[A,B] (xs : Stream[A ⇒ B], ys : Stream[A]) =
  (xs, ys) match {
    case (cons (f, fs), cons (s, ss)) ⇒
      cons (f (s), zapp (fs, ss))
    case (_, _) ⇒ Stream.empty
  }
```

Some example client code is given next:

```
def zipWith0 : Stream[Int] = zWith (Zero (), 0)
def map[A,B] (f : A ⇒ B) : Stream[A] ⇒ Stream[B] =
  zWith (Succ (Zero ()), f)
def zipWith3[A,B,C,D] (f : A ⇒ B ⇒ C ⇒ D) :
  Stream[A] ⇒ Stream[B] ⇒ Stream[C] ⇒ Stream[D] =
  zWith (Succ (Succ (Succ (Zero ()))), f)
```

6.5 ZipWith using prioritised overlapping implicits

Scala offers another solution for the n-ary *zipWith* problem, which avoids the explicit encoding of type-level natural numbers. This solution relies on Scala's support for disambiguating *overlapping implicits* by explicitly prioritizing the implicits. Although type classes in Haskell support *overlapping instances*, this solution is not directly applicable to Haskell.

```
trait ZipWith[S] {
  type ZipWithType
  def manyApp : Stream[S] ⇒ ZipWithType
  def zipWith : S ⇒ ZipWithType =
    f ⇒ manyApp (repeat (f))
}
class ZipWithDefault {
  implicit def ZeroZW[S] = new ZipWith[S] {
    type ZipWithType = Stream[S]
    def manyApp = xs ⇒ xs
  }
}
object ZipWith extends ZipWithDefault {
  def apply[S] (s : S) (implicit zw : ZipWith[S])
    : zw.ZipWithType = zw.zipWith (s)
  implicit def SuccZW[S,R]
    (implicit zw : ZipWith[R]) = new ZipWith[S ⇒ R] {
    type ZipWithType = Stream[S] ⇒ zw.ZipWithType
    def manyApp = xs ⇒ ss ⇒
      zw.manyApp (zapp (xs, ss))
  }
}
```

Figure 11. N-ary *zipWith* using prioritised implicits.

Figure 11 shows the alternative solution for the n-ary *zipWith* problem. Notably, the trait *ZipWith* and the methods *manyApp* and *zipWith* are not parameterised by natural numbers anymore.

When several implicit values are found for a certain type, disambiguation proceeds by the standard rules for static overloading. Finally, an additional tie-breaker rule is introduced that gives priority to implicits that are defined in a subclass over those in a parent class. That is why *SuccZW* will be preferred over *ZeroZW* in case of ambiguity.

For example, both *ZeroZW* and *SuccZW* are possible matches when an implicit of type $S \Rightarrow R$ is required. However, the ambiguity is resolved because *SuccZW* is defined in a subclass of *ZipWithDefault*, which defines *ZeroZW*.

Using this solution, definitions for *zipWith0* and *map* are:

```
def zipWith0 : Stream[Int] = ZipWith (0)
def map[A,B] (f : A ⇒ B) : Stream[A] ⇒ Stream[B] =
  ZipWith (f)
```

6.6 Encoding generalized constraints

Generalized constraints (Emir et al. 2006) provide a way to describe what it means for a type to be a subtype of

another type, and are used in practice in the Scala collection libraries (Odersky and Moors 2009). For example, a classic use-case for generalized constraints is the *flatten* method of a collection with elements of type T: it is only applicable if T is itself a collection. However, methods cannot constrain the type parameters of the class in which they are defined. We can represent this subtyping constraint indirectly as a function from T to the collection type $\mathit{Traversable}[U]$. This coercion can be seen as a witness to the subtype relation. Clearly, the caller of the method should not have to provide this witness explicitly.

```
sealed abstract class <:<[−S, +T] extends (S ⇒ T)
implicit def conforms[A] : A <:< A = new (A <:< A) {
  def apply (x : A) = x }
trait Traversable[T] {
  type Coll[X]
  def flatten[U] (implicit w : T <:< Traversable[U])
    : Coll[U]
}
```

To address this problem in Scala, we encode generalized constraints using the type constructor $<:<$. The trick is to use variance to extend the fact $A <:< A$ (witnessed by the implicit value $\mathit{conforms}[A]$) to the facts $S <: T$ for any S and T, where $S <: A$ and $A <: T$. According to the variance of the type constructor $<:<$, a value of type $A <: A$ can be used when a value of type $S <: T$ is expected. When type checking a call to *flatten* on a $\mathit{Traversable}[\mathit{List}[\mathit{Int}]]$, for example, an implicit of type $\mathit{List}[\mathit{Int}] <:< \mathit{Traversable}[?U]$ is searched, where $?U$ denotes the type variable that is used to infer the concrete type for U. Since $\mathit{conforms}[\mathit{List}[\mathit{Int}]]$ is the only solution, $?U$ is inferred to be *Int*. Moreover, as $<:<$ is a (higher-order) subtype of $\Rightarrow$, w is used as an implicit conversion in the body of *flatten* to convert expressions of type T to expressions of type $\mathit{Traversable}[U]$.

This example shows that functional dependencies (Jones 2000) are specified in Scala by the order of arguments and their grouping into argument lists. Type inference proceeds from left to right, with constraints being solved per argument list, so that, once a type variable is solved, the arguments in a later argument list have to abide. In implicit argument lists, the constraints that arise from the search for each implicit argument are solved immediately, so that implicit arguments must respect the instantiations of the type variables that result from these earlier arguments. Finally, as the implicit argument list must come last, implicit search cannot *actively* guide type inference in the explicit argument lists, although it can veto its results post factum.

6.7 Type theories using implicits

To conclude this section, we briefly discuss the pattern that underlies the previous examples: the idea of describing relations on types using implicits, which introduces a lightweight form of type-level computation. This pattern is applied in the Scala 2.8 collection library (Odersky and Moors 2009) to specify how transformations on collections affect the types of the elements and their containers.

Specifically, the $\mathit{CanBuildFrom}[\mathit{From}, \mathit{El}, \mathit{To}]$ relation is used to specify that the collection *To* can store elements of type *El* after transforming a collection of type *From*. This relation can be thought of as a function from the *types From* and *El* to the type of the derived collection *To*.

The crucial feature that makes defining these relations on types practical is the connection between implicit search and type inference. In fact, implicit search can be thought of as a generalization of type inference to values. Thus, a relation on types can be modeled by a type constructor of the same arity as the relation, where a tuple of types is considered in this relation if there is an implicit value of the type that results from applying the relation's type constructor to the concrete type arguments in the tuple.

To summarize, the examples described in this section are modeled using several type-level relations: the duality of two sessions; the relation between the argument and return type of the n-ary *zipWith* function; and $<:<$, the generalized constraint relation.

7. Discussion and Related Work

This section discusses the results of this paper and presents some related work. Also, an existing comparison between different languages in terms of their support for generic programming in the large (Siek and Lumsdaine 2008) is revised to include Scala and JavaGI. This comparison shows that Scala is very suitable for generic programming in the large.

7.1 Real-world applications

Implicits are widely used by Scala programmers. The largest real-world application of some of the techniques presented in this paper is probably the newly designed Scala collections library that ships as part of Scala 2.8. Odersky and Moors (2009) report their experiences in redesigning the Scala collections and argue that implicits, and their ability to specify piece-wise type-level functions, play a crucial role in their design.

In retrospect, the results reported by Odersky and Moors are not too surprising. The C++ generic programming community has long learned to appreciate the value of associated types to define such piece-wise functions for collection types. The developments presented in Section 6 show how associated types can be more naturally defined using type members and dependent method types. The standard template library (STL) (Musser and Saini 1995) and Boost (Boost) libraries are prime examples of generic programming with C++ templates. In some sense, Scala's 2.8 collections can be viewed as the STL/Boost of Scala.

In the functional programming communities the term "generic programming" is often used to mean *datatype-*

generic programming (DGP) (Gibbons 2003). DGP can be viewed as an advanced form of generic programming in which the structure of types is used to derive generic algorithms. Oliveira and Gibbons (2008) show that Scala is particularly well-suited for DGP and that it is, in some ways, better than Haskell. This is partly due to the flexibility of implicits in comparison with type classes.

7.2 Type classes, JavaGI and concepts

Type classes Haskell type classes were originally designed as a solution for ad-hoc polymorphism (Hall et al. 1996; Wadler and Blott 1989). Many languages have mechanisms inspired by type classes (Gregor et al. 2006; Haftmann and Wenzel 2006; Shapiro et al. 2008; Sozeau and Oury 2008; Wehr 2009), and implicits are no exception. Implicits are the minimal delta needed to enable type class programming in an OOPL with support for parametric polymorphism. This extension can be seen as untangling type classes into the (existing) OO class system, and a mechanism for type-directed implicit parameter passing. The most obvious downside of the untangling approach is that implicit values and implicit arguments must play by the rules that also govern normal values and arguments: that is, they must be named. However, these names are useful when disambiguation is needed, and, at least for implicit arguments, they can be eschewed using context bounds. Finally, the OO philosophy of limiting type inference to the inside of encapsulation boundaries entails that context bounds are not inferred, unlike in Haskell.

Nevertheless, this untangling has several benefits. Implicit arguments may still be passed explicitly when necessary, while Haskell requires duplication when the dictionary needs to be passed explicitly. For example a *sort* function (in which the ordering dictionary is passed implicitly), and a *sortBy* function (in which the ordering dictionary is passed explicitly) are needed in Haskell. Furthermore, since a type class is encoded as a first-class type, the language's full range of polymorphism applies. In Haskell, it is not possible to directly abstract over a type class (Hughes 1999).

Several authors noted that Haskell type classes can be limiting for certain applications due to the impossibility of explicitly passing arguments and abstracting over type classes (Dijkstra and Swierstra 2005; Hughes 1999; Kahl and Scheffczyk 2001; Lämmel and Jones 2005; Oliveira and Gibbons 2008; Oliveira and Sulzmann 2008). Thus, there have been a number of proposals aimed at lifting some of these restrictions (Dijkstra and Swierstra 2005; Kahl and Scheffczyk 2001; Orchard and Schrijvers 2010). However, none of these proposals has been adopted in Haskell. There is also a proposal for an ML-module system that allows modules to be implicitly passed based on their types, thus allowing many typical type class programs to be recovered by suitably marking module parameters as implict (Dreyer et al. 2007).

Type class extensions such as functional dependencies (Jones 2000), associated types (Chakravarty et al. 2005b), associated datatypes (Chakravarty et al. 2005a), and type-families (Schrijvers et al. 2008) provide simple forms of type-level computation. This work shows that by combining type members and dependent method types it is possible to encode associated types, which are a fundamental mechanism for supporting advanced forms of generic programming (Garcia et al. 2007). Furthermore, prioritized overlapping implicits allow the definition of type-level functions with a default, catch-all case, that can be overridden by other cases in subclasses. This functionality provides a limited form of *concept-based overloading* (Siek and Lumsdaine 2008). In Haskell such overlapping definitions are forbidden for associated types and type-families (Chakravarty et al. 2005b; Schrijvers et al. 2008); and have limited applicability with functional dependencies. Type-families allow a natural definition of type-level functions, whereas type members and dependent method types can only express such definitions indirectly. It would be interesting to explore a mechanism similar to type-families in the context of OO languages.

JavaGI Inspired by Lämmel and Ostermann (2006), Wehr (2009) proposes JavaGI: a variant of Java aimed at bringing the benefits of type classes into Java. JavaGI provides an alternative viewpoint in the design space of type-class-like mechanisms in OO languages. JavaGI has *generalized interfaces* and *generalized interface implementations* in addition to conventional OO interfaces and classes. Scala, on the other hand, models type-class-like mechanisms using the conventional OO class system. JavaGI supports a form of *dynamic multiple dispatching* in a similar style to *multi-methods* (Chambers and Leavens 1995; Clifton et al. 2000). This allows JavaGI to express *open classes* (Clifton et al. 2000) through retroactive implementations. Scala is less expressive in this respect as it does not support dynamic multiple dispatching; only static multiple dispatching is supported. However, Scala supports fully modular typechecking, which is not the case for JavaGI.

Concepts *Concepts* (Musser and Saini 1995) are an important notion for C++ style generic programming. They describe a set of requirements for the type parameters used by a generic algorithm. Although concepts are widely used by the C++ community to document generic algorithms, current versions of C++ have no representation for concepts within the C++ language. Nevertheless the situation is likely to change. Gregor et al. (2006) proposed linguistic support for concepts in C++. Moreover, motivated by the lack of modular typechecking and separate compilation in C++, concepts have been studied in the F^G calculus (Siek and Lumsdaine 2005) and the *G* language (Siek and Lumsdaine 2008).

Concepts are closely related to type classes and there is a fairly accepted idea that type classes can be viewed as an alternative language mechanism to express concepts (Bernardy et al. 2008). Indeed, in several comparative studies, type classes score very well when it comes to language sup-

	C++	*SML*	*OCaml*	*Haskell*	*Java*	*C#*	*Cecil*	*C++0X*	*G*	*JavaGI*	*Scala*
Multi-type concepts	-	●	○	●	●[2]	●[2]	◐	●	●	●	●[2]
Multiple constraints	-	◐	◐	●	●	●	●	●	●	●	●
Associated type access	●	●	◐	●	◐	◐	◐	●	●	◐	●[1]
Constraints on assoc. types	-	●	●	●	◐	◐	●	●	●	◐	●[1]
Retroactive modeling	-	●	●	●	◐[2]	◐[2]	●	●	●	●	●[23]
Type aliases	●	●	●	●	○	○	○	●	●	○	●
Separate compilation	○	●	●	●	●	●	◐	○	●	●	●
Implicit arg. deduction	●	○	●	●	◐[5]	◐[5]	◐	●	●	●	●[3]
Modular type checking	○	●	◐	●	●	●	◐	◐	●	◐	●
Lexically scoped models	○	●	○	○	○	○	○	○	●	○	●
Concept-based overloading	●	○	○	○	○	○	●	●	◐	○	◐[4]
Equality constraints	-	●	○	●	○	○	○	●	●	○	●
First-class functions	○	●	●	●	○	◐	●	●	◐	○	●

Figure 12. Level of support for generic programming in several languages. Key: ●='good', ◐='sufficient', ○='poor' support. The rating "-" in the C++ column indicates that while C++ does not explicitly support the feature, one can still program as if the feature were supported. Notes: 1) supported via type members and dependent method types 2) supported via the CONCEPT pattern 3) supported via implicits 4) partially supported by prioritized overlapping implicits 5) decreased score due to the use of the CONCEPT pattern

port for generic programming (Garcia et al. 2007; Siek and Lumsdaine 2008) (see also Figure 12). However, there are some differences in purpose between type classes and implicits, and concepts in C++. In C++ performance is considered as a critical requirement, and the template mechanism is designed so that, at *compile-time*, templated code is specialized. Thus a potential mechanism for expressing concepts in C++ should not jeopardize these performance benefits (Dos Reis and Stroustrup 2006). In contrast, the main motivation for type classes and implicits is abstraction and convenience, providing additional flexibility through indirection, at the potential cost of run-time performance. Although Scala already supports some *user-driven type specialization* (Dragos and Odersky 2009), further work needs to be done to investigate whether implicits could be adapted to a system supporting full compile-time specialization.

The CONCEPT pattern is aimed at expressing concepts using a standard OO class system without the performance constraints of C++. In an OO language like Scala, the CONCEPT pattern, combined with implicits and support for associated types through type members and dependent method types, provides an effective platform for generic programming in the large.

7.3 Generic programming in the large

In studies by Garcia et al. (2007); Siek and Lumsdaine (2008), support for generic programming in several different languages is investigated, with particular emphasis on how such languages can model concepts. Figure 12 shows the level of support for generic programming in various languages. For the most part the scores are inherited from the study presented by Siek and Lumsdaine (2008). We added the scores for JavaGI and Scala and adjusted a couple of scores in Java and C#. The scores for JavaGI are derived from the related work discussions by Wehr (2009); we did not do any experiments with JavaGI. These scores and adjustments are discussed next.

Adjustments on Java and C# scores In the original comparison, Siek and Lumsdaine (2008) give Java and C# bad scores at both multi-type concepts and retroactive modeling. The main reason for those scores is that concepts are modeled by a conventional subtyping hierarchy. For example, the *Comparable* concept and the corresponding *Apple* model, are implemented as follows:

```
trait Comparable[T] {
  def better(x:T):Boolean
}
public class Apple extends Comparable[Apple] {...}
```

As discussed in Section 4 this solution is problematic for supporting retroactive modeling and multi-type concepts. In contrast, if the CONCEPT is used, models can be added retroactively and multi-type concepts can be expressed conveniently. The drawback of the CONCEPT solution in languages like Java or C#, as also discussed in Section 4, is that there is some additional overhead to use conceptual methods. This is reflected in the score for retroactive modeling, which is only sufficient, since support for this feature is not as natural as it could be. Also, the score for implicit argument deduction is affected by the use of the CONCEPT pattern because the concept constraints have to be passed explicitly. In Java and C#, the original solution using bounded

polymorphism is better in this respect because no such overhead exists.

JavaGI Since JavaGI is a superset of Java, it inherits most of its scores. In JavaGI retroactive modeling and multi-type concepts are naturally supported through generalized interfaces. Wehr (2009) explicitly states that lexically scoped models and concept-based overloading are not supported. Furthermore, although type-checking is mostly modular, a final global check is necessary. Thus JavaGI only partially supports modular type-checking.

Scala Using the CONCEPT pattern we can model multi-type concepts, multiple constraints and support retroactive modeling. Furthermore, Scala's support for implicits means that the drawbacks of the Java and C# solutions in terms of the additional overhead, do not apply to Scala. Thus, Scala scores well in both the implicit argument deduction and the retroactive modeling criteria. Section 6 shows that associated types are supported in Scala through type members and dependent method types, and type members can also be used as *type aliases*. As shown in Section 3, Scala supports lexically scoped models. Furthermore type-checking is fully modular. Prioritized overlapping implicits provide some support for concept-based overloading as illustrated by the *zipWithN* example in Section 6.5. However, overlapping models have to be structured using a subtyping hierarchy, which may not always be desirable. Thus, the score for this feature is only sufficient. Finally, Scala has full support for first-class functions and it also supports equality constraints.

In summary Scala turns out to be a language with excellent support for generic programming features, managing to fare at the same level, or even slightly better, than G (which was specially designed as a language for generic programming in the large) or Haskell (which has been recognized has having very good support for generic programming).

8. Conclusion

This paper shows that it is possible to have the benefits of type classes in a standard OO language with generics, by using the CONCEPT pattern to express type-class style programs. However some convenience is lost, especially for traditional applications aimed at using type classes for constrained polymorphism.

Implicits are a modest extension that can be added to statically typed languages. Implicits bring back the convenience of use of type classes, but they have wider applicability and are useful in several other domains. With the improved support for type-inference that we are already seeing in mainstream languages like Java or C#, it is only natural to expect that implicits will eventually find their way into those languages.

Type members and dependent method types add extra power to the language and a combination of the two mechanisms allows associated types to be expressed. In combination with implicits, type members and dependent method types make Scala an language ready for generic programming in the large.

Acknowledgments

This work benefited from several discussions that we had with William Cook. We are grateful to Tom Schrijvers, Jonathan Shapiro and Marcin Zalewski for their useful comments. Bruno Oliveira was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / Korea Science and Engineering Foundation (KOSEF) grant number R11-2008-007-01002-0.

References

J. P. Bernardy, P. Jansson, M. Zalewski, S. Schupp, and A. Priesnitz. A comparison of C++ concepts and Haskell type classes. In *WGP '08*, pages 37–48, 2008.

Boost. The Boost C++ libraries. http://www.boost.org/, 2010.

K. Bruce, L. Cardelli, G. Castagna, G. T. Leavens, and B. Pierce. On binary methods. *Theor. Pract. Object Syst.*, 1(3):221–242, 1995.

P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA '89*, pages 273–280, 1989.

M. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. pages 1–13, 2005a.

M. Chakravarty, G. Keller, and Simon Peyton Jones. Associated type synonyms. In *ICFP '05*, pages 241–253, 2005b.

C. Chambers and G. T. Leavens. Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, 1995.

C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00*, pages 130–145, 2000.

W. R. Cook. On understanding data abstraction, revisited. *SIGPLAN Not.*, 44(10):557–572, 2009.

W. R. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Inf. Comput.*, 114(2):329–350, 1994.

O. Danvy. Functional unparsing. *J. Funct. Program.*, 8(6):621–625, 1998.

A. Dijkstra and S. D. Swierstra. Making implicit parameters explicit. Technical report, Institute of Information and Computing Sciences, Utrecht University, 2005. URL `http://www.cs.uu.nl/research/techreps/UU-CS-2005-032.html`.

G. Dos Reis and B. Stroustrup. Specifying C++ concepts. In *POPL '06*, pages 295–308, 2006.

I. Dragos and M. Odersky. Compiling generics through user-directed type specialization. In *ICOOOLPS '09*, pages 42–47, 2009.

D. Dreyer, R. Harper, M. M. T. Chakravarty, and G. Keller. Modular type classes. In *POPL '07*, pages 63–70, 2007.

B. Emir, A. Kennedy, C. V. Russo, and D. Yu. Variance and generalized constraints for C$^\#$ generics. In *ECOOP*, pages 279–303, 2006.

R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17(2):145–205, 2007.

J. Gibbons. Patterns in datatype-generic programming. In *The Fun of Programming, Cornerstones in Computing*, pages 41–60. Palgrave, 2003.

D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA '06*, pages 291–310, 2006.

F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In *TYPES*, pages 160–174, 2006.

C. V. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2): 109–138, 1996.

K. Honda. Types for dynamic interaction. In *CONCUR '93*, pages 509–523, 1993.

J. Hughes. Restricted data types in Haskell. In *Haskell Workshop*, 1999.

J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. An analysis of constrained polymorphism for generic programming. In *MPOOL '03*, page 87–107, 2003.

M. P. Jones. Type classes with functional dependencies. In *ESOP '00*, pages 230–244, 2000.

W. Kahl and J. Scheffczyk. Named instances for Haskell type classes. In *Haskell Workshop*, 2001.

A. Kennedy and C. V. Russo. Generalized algebraic data types and object-oriented programming. *OOPSLA '05*, pages 21–40, 2005.

O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04*, pages 96–107, 2004.

O. Kiselyov, S. Peyton Jones, and C. Shan. Fun with type functions, 2009. URL `http://research.microsoft.com/en-us/um/people/simonpj/papers/assoc-types/`.

R. Lämmel and S. P. Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP '05*, pages 204–215, 2005.

R. Lämmel and K. Ostermann. Software extension and integration with type classes. In *GPCE '06*, pages 161–170, 2006.

D. B. MacQueen. Modules for Standard ML. In *LISP and Functional Programming*, pages 198–207, 1984.

C. McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.*, 12(5):375–392, 2002.

J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. In *OOPSLA '08*, pages 423–438, 2008.

D. Musser and A. A. Stepanov. Generic programming. In *Symbolic and algebraic computation: ISSAC 88*, pages 13–25. Springer, 1988.

D. R. Musser and A. Saini. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

M. Odersky, 2006. URL `http://www.artima.com/weblogs/viewpost.jsp?thread=179766`.

M. Odersky. *The Scala Language Specification, Version 2.8*. EPFL, 2010. URL `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.

M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In *FSTTCS*, pages 427–451, 2009.

M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *ECOOP03*, pages 201–224. Springer-Verlag, 2003.

M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, EPFL, 2006.

B. C. d. S. Oliveira and J. Gibbons. Scala for generic programmers. In *WGP '08*, pages 25–36, 2008.

B. C. d. S. Oliveira and M. Sulzmann. Objects to unify type classes and GADTs. URL `http://www.comlab.ox.ac.uk/people/Bruno.Oliveira/objects.pdf`. April 2008.

D. Orchard and T. Schrijvers. Haskell type constraints unleashed. In *FLOPS '10*. Springer-Verlag, 2010.

S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

S. Peyton Jones, M. Jones, and E. Meijer. Type classes: exploring the design space. In *Haskell Workshop*, 1997.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP '06*, pages 50–61, 2006.

S. Peyton Jones, S. Marlow, et al. The Glasgow Haskell Compiler, 2009. URL `http://www.haskell.org/ghc/`.

T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *ICFP '08*, pages 51–62, 2008.

J. S. Shapiro, S. Sridhar, and M. S. Doerrie. BitC language specification, 2008. URL `http://www.coyotos.org/docs/bitc/spec.html`.

J. G. Siek and A. Lumsdaine. Essential language support for generic programming. In *PLDI '05*, pages 73–84, 2005.

J. G. Siek and A. Lumsdaine. A language for generic programming in the large. *Science of Computer Programming*, In Press, Corrected Proof, 2008. URL `http://www.sciencedirect.com/science/article/B6V17-4TJ6F7D-1/2/7d624b842e8dd84e792995d3422aee21`.

M. Sozeau and N. Oury. First-class type classes. In *TPHOLs '08*, pages 278–293, 2008.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, pages 60–76, 1989.

S. Wehr. *JavaGI: A Language with Generalized Interfaces*. PhD thesis, University of Freiburg, Department of Computer Science, December 2009.

```
class Ord a where
  eq      :: a → a → Bool
  compare :: a → a → Bool
instance Ord Int where
  eq a b      = compare a b ∧ compare b a
  compare x y = x ⩽ y
instance Ord a ⇒ Ord [a] where
  eq a b = compare a b ∧ compare b a
  compare l1 l2 = case (l1, l2) of
    (x : xs, y : ys) → if (eq x y) then compare xs ys
      else compare x y
    (_, [])  → False
    (_, _)   → True
```

Figure 13. Ordering concept in Haskell (cf. Figure 3)

```
class Set s where
  empty    :: s
  insert   :: s → Int → s
  contains :: s → Int → Bool
  union    :: s → s → s
instance Set [Int] where
  empty    = []
  insert   = λx y → y : x
  contains = λx y → elem y x
  union    = λx y → List.union x y
instance Set (Int → Bool) where
  empty    = λx → False
  insert   = λf y z → y ≡ z ∧ f z
  contains = λf y → f y
  union    = λf g y → f y ∨ g y
```

Figure 14. A Set ADT using Type Classes (cf. Figure 5)

A. Type Class Examples in Haskell

This section shows the Haskell versions of the programs used in this paper.

```
class Format a where
  format :: String → a
printf :: Format a ⇒ a
printf = format ""
instance Format a ⇒ Format (Int → a) where    -- I
  format s = λi → format (s ++ show i)
instance Format a ⇒ Format (Char → a) where    -- C
  format s = λc → format (s ++ show c)
instance Format String where    -- E
  format s = s
```

Figure 15. Printf using Type Classes (cf. Figure 6)

```
class Session a where
  type Dual a
  run :: a → Dual a → IO ()
instance (Session b) ⇒ Session (In a b) where
  type Dual (In a b) = Out a (Dual b)
  run (In f) (Out a d) = f a >>= λb → d >>= λc → run b c
instance (Session b) ⇒ Session (Out a b) where
  type Dual (Out a b) = In a (Dual b)
  run (Out a d) (In f) = f a >>= λb → d >>= λc → run c b
instance Session Stop where
  type Dual Stop = Stop
  run Done Done = return ()
```

Figure 16. Session types (cf. Figure 9)

```
data Zero = Zero
data Succ n = Succ n
class ZipWith n s where
  type ZipWithType n s
  manyApp :: n → [s] → ZipWithType n s
  zipWithN :: n → s → ZipWithType n s
  zipWithN n f = manyApp n (repeat f)
instance ZipWith Zero t where
  type ZipWithType Zero t = [t]
  manyApp Zero fs = fs
instance ZipWith n u ⇒ ZipWith (Succ n) (s → u) where
  type ZipWithType (Succ n) (s → u) = [s] → ZipWithType n u
  manyApp (Succ n) fs = λss → manyApp n (fs <<< ss)
```

Figure 17. N-Ary zipWith using associated types (cf. Figure 10)