Scala for Generic Programmers

Bruno C. d. S. Oliveira and Jeremy Gibbons

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{bruno,jg}@comlab.ox.ac.uk

Abstract

Datatype-generic programming involves parametrization by the shape of data, in the form of type constructors such as 'list of'. Most approaches to datatype-generic programming are developed in the lazy functional programming language Haskell. We argue that the functional object-oriented language Scala is in many ways a better setting. Not only does Scala provide equivalents of all the necessary functional programming features (such parametric polymorphism, higher-order functions, higher-kinded type operations, and type- and constructor-classes), but it also provides the most useful features of object-oriented languages (such as subtyping, overriding, traditional single inheritance, and multiple inheritance in the form of traits). We show how this combination of features benefits datatype-generic programming, using three different approaches as illustrations.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Datatype-Generic Programming, Polytypic Programming, Scala

1. Introduction

Datatype-generic programming is about writing programs that are parametrized by a datatype, such as lists or trees. This is different from parametric polymorphism, or 'generics' as the term is used by most object-oriented programmers: parametric polymorphism abstracts from the 'integers' in 'list of integers', whereas datatype-generic programming abstracts from the 'list of'.

There is a large and growing collection of techniques for writing datatype-generic programs. These range from domain-specific languages such as Generic Haskell [Hinze and Jeuring, 2002] and Charity [Cockett and Fukushima, 1992], through language extensions such as Scrap Your Boilerplate [Lämmel and Peyton Jones, 2003] and Template Haskell [Sheard and Peyton Jones, 2002], to libraries for existing general purpose languages such as Generics for the Masses [Hinze, 2006] and Adaptive Object-Oriented Programming [Lieberherr, 1996]. Evidently datatype-generic programming is quite a hot topic.

WGP'08 September 20, 2008, Victoria, BC, Canada.

Despite the rather wide variety of host languages involved in the techniques listed above, the casual observer might be forgiven for thinking that 'Haskell is the programming language of choice for discriminating datatype-generic programmers' (to paraphrase the first prize of the ICFP Programming Contest). Our purpose in this paper is to argue to the contrary; we believe that although Haskell is 'a fine tool for many datatype-generic applications', it is not necessarily the best choice for them.

Specifically, we argue that the discriminating datatype-generic programmer ought seriously to consider using Scala, a relatively recent language providing a smooth integration of the functional and object-oriented paradigms. Scala offers equivalents of all the familiar features cherished by datatype-generic Haskell programmers, such parametric polymorphism, higher-order functions, higher-kinded type operations, and type- and constructor-classes. (The most significant missing feature is lazy evaluation.) But in addition, it offers some of the most useful features of object-oriented programming languages, such as subtyping, overriding, and both single and a form of multiple inheritance. We show that not only can Haskell techniques for generic programming be conveniently replicated in Scala, but that the extra expressivity provides some important additional benefits in extensibility and reuse.

We are not the first to associate datatype-generic programming with Scala. Indeed, at the previous Workshop on Generic Programming, Moors et al. [2006] presented a translation into Scala of a Haskell library of 'origami operators' [Gibbons, 2006]; we discuss this translation in depth in Section 4. And of course, it is not really surprising that Scala should turn out to subsume Haskell, since its principal designer is a well-known functional programmer.

We feel that our main contribution is as a position paper: a call to datatype-generic programmers to look beyond Haskell, and particularly to look at Scala; not only is Scala good enough, in some ways it is better than Haskell for datatype-generic programming. Some of those advantages derive from Scala's mixed-paradigm nature, and do not translate back into Haskell; but others (such as case classes and anonymous case analyses, as we shall see) would fit perfectly well into Haskell. So we are not just trying to take programmers from the Haskell camp; we are also trying to add features to the Haskell language.

As a secondary contribution, we show that Scala is more of a functional programming language than is typically appreciated. We believe that Scala tends to be seen only as an object-oriented language that happens to have some functional features, and so potential users feel that they have to use it in an object-oriented way. (For example, Moors et al. [2006] claimed to be 'staying as close to the original work as possible' in their translation, but as we show in Section 4 they still ended up less functional than they might have done.) Scala is also a functional programming language that happens to have object-oriented features. Indeed, it offers the best of both worlds, and this paper is also a tutorial in getting the best out of Scala as a multi-paradigm language.

The rest of this paper is structured as follows. Sections 2 and 3 introduce the basics of Scala, and those more advanced features of its type and class system on which we depend. Our contribution starts in Section 4, in which we present an alternative to (and, we argue, improvement of) Moors et al.'s encoding of the origami operators. Section 5 discusses the extensibility benefits of using open datatypes for type representations, which can serve as a basis for generic programming libraries [Cheney and Hinze, 2002, Weirich, 2006]. Section 6 discusses generic programming approaches based on type classes, such as *Generics for the Masses* [Hinze, 2006], and explains how the reuse benefits from object-oriented features, namely inheritance and overriding, can be helpful. (In this paper we shall not present the full code required to run the examples for reasons of space, but Oliveira [2008] presents the complete Scala code for all three approaches.) Section 7 concludes.

2. The Basics of Scala

Scala is a strongly typed programming language that combines object-oriented and functional programming features. Although inspired by recent research, Scala is not just a research language; it is also aimed at industrial usage: a key design goal of Scala is that it should be easy to interoperate with mainstream languages like Java and C#, making their many libraries readily available to Scala programmers. The user base of Scala is already quite significant, with the compiler being actively developed and maintained. For a more complete introduction to and description of Scala, see Odersky [2006a, 2007a,b], Schinz [2007].

2.1 Definitions and values

Functions are defined using the **def** keyword. For example, the squaring function on *Double*s could be written:

def *square* $(x : Double) : Double = x * x$

Scala distinguishes between definitions and values. In a definition **def** $x = e$, the expression e will not be evaluated until x is used. Scala also offers a value definition **val** $x = e$, in which the right-hand side e is evaluated at the point of definition. However, only definitions can take parameters; values must be constants.

2.2 First-class functions

Functions in Scala are first-class values, so *higher-order functions* are supported. For example, to define the function *twice* that applies given a function f twice to its argument x, we could write:

def *twice* $(f : Int \Rightarrow Int, x : Int) : Int = f\ (f\ (x))$

Scala supports *anonymous functions*. For instance, to define a function that raises an integer to the fourth power, we could use the function *twice* together with an anonymous function:

def *power* $(x : Int) : Int = twice\ ((y : Int) \Rightarrow y * y, x)$

The first argument of the function *twice* is an anonymous function that takes an integer y and returns another integer $y * y$.

Scala also supports *currying*. To declare a curried version of *twice*, we can write:

def *curryTwice* $(f : Int \Rightarrow Int)\ (x : Int) : Int = f\ (f\ (x))$

2.3 Parametric polymorphism

Like Haskell and ML (and more recently Java and C#), Scala supports *parametric polymorphism* (known as *generics* in the object-oriented world). For example, function composition can be defined as follows:

def *comp* $[a, b, c]\ (f : b \Rightarrow c)\ (g : a \Rightarrow b)\ (x : a) : c = f\ (g\ (x))$

2.4 Call-by-name arguments

Function arguments are, by default, passed *by value*, being evaluated at the point of function application. This gives Scala a strict functional programming flavour. However, we can also pass arguments *by name*, by prefixing the type of the argument with '⇒'; the argument is then evaluated at each use within the function definition. This can be used to emulate lazy functional programming; although multiple uses do not share evaluation, it is still useful, for example, for defining new control structures. Scala's parser combinators are a good example of the use of laziness: the combinator *Then* tries to apply a parser p, and if that parser succeeds, applies another parser q to the remainder of the input:

def *Then* $(p : Parser)\ (q : \Rightarrow Parser) : Parser = \ldots$

Here, the second parser q is lazy: only if q is needed will it be evaluated.

2.5 Type inference

The design goal of interoperability with languages like Java requires Scala's type system compatibility. In particular, this means that Scala needs to support subtyping and (name-) overloaded definitions such as:

def *add* $(x : Int) : Unit = \ldots$
def *add* $(x : String) : Unit = \ldots$

This makes type inference difficult. Nevertheless, Scala does support some type inference. In particular, it is possible most of the time to infer the return type of a definition and the type of a lambda variable. For example, an alternative definition to *power* would be:

def *power* $(x : Int) = twice\ (y \Rightarrow y * y, x)$

in which both the return type and the type of the lambda variable y are inferred.

3. Scala's Object System

Scala has a rich object system, including object-oriented constructions familiar from mainstream languages like Java or C# such as classes, abstract classes, subtyping and inheritance. Scala also incorporates some less common concepts. In particular, there is a concrete notion of *object*, and interfaces are replaced by the more general notion of *traits* [Schärli et al., 2003], which can be composed using a form of mixin composition. Furthermore, Scala introduces the notion of *case classes*, whose instances can be decomposed using case analysis and pattern matching.

In this section, we introduce a subset of the full Scala object system. We believe, however, that this subset is particularly expressive, and mostly subsumes the other features. In our opinion, the object system is one of the rough edges of Scala, having many different (and largely overlapping) constructs that entail significant complexity in the language; we believe that there is potential for simplification in this area. Having said that, the fact is that Scala's object system is very powerful, precluding the need for a separate module system and enjoying of a form of multiple inheritance via the use of traits and mixin composition.

3.1 Traits and mixin composition

Instead of interfaces, Scala has the more general concept of *traits* [Schärli et al., 2003]. Like interfaces, traits can be used to define *abstract methods* (that is, method signatures). However, unlike interfaces, traits can also define concrete methods. Traits can be combined using *mixin composition*, making a safe form of *multiple inheritance* possible, as the following example demonstrates:

```
trait Hello {
  val hello          = "Hello!"
}
```

```
trait List[A]
case class Nil[A] extends List[A]
case class Cons[A] (x:A, xs:List[A]) extends List[A]

def len[a] (l:List[a]):Int = l match {
  case Nil()          ⇒ 0
  case Cons(x, xs) ⇒ 1 + len(xs)
}

def ins[a <: Ordered[a]] (x:a, l:List[a]):List[a] =
  l match {
    case Nil()          ⇒ Cons(x, Nil[a])
    case Cons(y, ys) ⇒ if (x ⩽ y) Cons(x, Cons(y, ys))
                       else Cons(y, ins(x, ys))
  }
```

Figure 1. Algebraic datatypes and case analysis in Scala.

```
trait HowAreU {
  val howAreU       = "How are you?"
}
trait WhatIsUrName {
  val whatIsUrName = "What is your name?"
}
trait Shout {
  def shout (str:String):String
}
```

In this example, we use traits in much the same way as we would have used classes, allowing the declaration of both abstract methods like *shout* and concrete methods like *hello*, *howAreU* and *whatIsUrName*. In a single-inheritance language like Java or C#, it would not be possible to define a subclass that combined the functionality of the four code blocks above. However, mixin composition allows any number of traits to be combined:

```
trait Basics extends Hello with HowAreU
             with WhatIsUrName with Shout {
  val greet                = hello + "  " + howAreU
  def shout (str:String) = str.toUpperCase()
}
```

The trait *Basics* inherits methods from *Hello*, *HowAreU* and *WhatIsUrName*, implements the method *shout* from *Shout*, and defines a value *greet* using the inherited methods *hello* and *howAreU*.

3.2 Objects and case classes

New object instances can be created as in conventional object-oriented languages by using the **new** construct. For example, we could have defined a new *Basics* object by:

```
def basics1 = new Basics() {}
```

Alternatively, Scala supports a distinct notion of **object**:

```
object basics2 extends Basics
```

While the self-type of $basics_1$ is *Basics*, the self-type of $basics_2$ is a new anonymous subtype of *Basics*; this small difference means that there are situations where the two are not interchangeable.

Scala also supports the notion of *case classes*, which simplify the definition of functions by case analysis. In particular, they allow the emulation of algebraic datatypes from conventional functional languages. Figure 1 gives definitions for the equivalent to the algebraic datatype of lists and the length and (ordered) insertion functions. The trait *List*[*A*] declares the type of lists parametrized by some element type *A*; the case classes *Nil* and *Cons* act as the two constructors of lists. The function *len* is defined using standard case analysis on the list value *l*. The function *ins* shows another definition by case analysis on lists, and also demonstrates the use of *type-parameter bounds*: the list elements must be ordered.

Case classes do not require the use of the **new** keyword for instantiation, as they provide a more compact syntax inspired by functional programming languages:

```
val alist = Cons(3, Cons(2, Cons(1, Nil())))
```

3.3 Higher-kinded types

Type-constructor polymorphism and constructor (type) classes have proved themselves very useful in Haskell allowing, among other things, the definition of concepts such as monads [Wadler, 1993], applicative functors [McBride and Paterson, 2007], and container-like abstractions. This motivated the recent addition of type-constructor polymorphism to Scala [Moors et al., 2008]. For example, a very simple interface for the *Iterable* class could be defined in Scala as:

```
trait Iterable[A, Container[_]] {
  def map[B] (f:A ⇒ B):Container[B]
  def filter (p:A ⇒ Boolean):Container[A]
}
```

Note that *Iterable* is parametrized by *Container*[_], which is a type that is itself parametrized by another type — in other words, a type constructor. By parametrizing over the type constructor *Container* rather than a particular type *Container*[*A*], we can use the parameter in method definitions with different types. In particular, in the definition of *map*, the return type is *Container*[*B*], where *B* is a type parameter of the method *map*; with parametrization by types only, we would have to content ourselves with a homogeneous *map*.

3.4 Abstract types

Scala has a notion of *abstract types*, which provide a flexible way to abstract over concrete types used inside a class or trait declaration. Abstract types are used to hide information about internals of a component, in a way similar to their use in SML [Harper and Lillibridge, 1994] and OCaml [Leroy, 1994]. As with any other kind of class member, abstract types in a class must be given concrete definitions before the class can be instantiated. Odersky and Zenger [2005] argue that abstract types are essential for the construction of reusable components: they allow information hiding over several objects, which is a key part of component-oriented programming.

Figure 2 shows a typical example of an ML-style abstract datatype for sets. The abstract trait *SetInterface* declares the types and the operations required by sets. The abstract types *A* and *Set* (which is a type constructor) are, respectively, abstractions over the element type and the shape of the set. The operations supported by the set interface are *empty*, *insert* and *extract*. The trait *SetOrdered* presents a concrete refinement of *SetInterface*, in which sets are implemented with lists and the elements of the set are ordered.

3.5 Implicit parameters and type classes

Scala's *implicit parameters* allow some parameters to be inferred implicitly by the compiler on the basis of type information; as noted by Odersky [2006b], they can be used to emulate Haskell's type classes [Hall et al., 1996]. Consider this approximation to the concept of a monoid, omitting any formalization of the monoid laws [Odersky, 2006a]:

```
trait Monoid[a] {
  def unit           :a   // unit of add
  def add (x:a, y:a):a    // associative
}
```

This is clearly analogous to a type class. An example object would be a monoid on strings, with the unit being the empty string and addition being the concatenation of strings.

```
trait SetInterface {
  type Set[_]
  type A
  def empty : Set[A]
  def insert (x:A, q:Set[A]) : Set[A]
  def extract (q:Set[A]) : Option[(A, Set[A])]
}
trait SetOrdered extends SetInterface {
  type Set[X] = List[X]
  type A <: Ordered[A]
  def empty = Nil()
  def insert (x:A, q:Set[A]) = ins(x, q)
  def extract (q:Set[A]) = q match {
      case Nil()          ⇒ None
      case Cons(x, xs) ⇒ Some(x, xs)
    }
}
```

Figure 2. An abstract datatype for sets.

```
implicit object strMonoid extends Monoid[String] {
  def unit              = ""
  def add (x:String, y:String) = x.concat(y)
}
```

Again, there is a clear correspondence with an instance declaration in Haskell. Ignoring the **implicit** keyword for a moment, we can now define operations that are generic in the monoid:

```
def sum[a] (xs:List[a]) (implicit m:Monoid[a]) : a =
  if (xs.isEmpty) m.unit
  else m.add (xs.head, sum(xs.tail)(m))
```

We can now use *sum* in the following way (as we would have done normally):

```
def test1 = sum (List ("a", "bc", "def")) (strMonoid)
```

However, we can omit the second argument to *sum*, since the compiler has enough information to infer it automatically:

```
def test2 : String = sum (List ("a", "bc", "def"))
```

This works because (a) the **implicit** keyword in the object states that *strMonoid* is the default value for the type *Monoid*[*String*], and (b), the **implicit** keyword in the definition of *sum* states that the argument *m* may be omitted if there exists an implicit object in scope with the type of *Monoid*[*a*]. (If there are multiple such objects, the most specific one is chosen.) The second use of *sum*, with the implicit parameter inferred by the compiler, is similar to Haskell usage; however, it is more flexible, because it also provides the option to provide an explicit value overriding the one implied by the type.

4. Scala as a DGP Language

In order to support truly datatype-generic programming, languages should support three forms of parametrization: "by the *shape* of the computation, which is determined by the shape of the underlying data, and represented by a type constructor (an operation on types); by the *element type* (a type); and by the *body* of the computation, which is a higher-order argument (a value, typically a function)." [Gibbons, 2006]. As we have seen, Scala readily supports all of these parametrization forms: parametrization by type is provided by generics; parametrization by computation comes from higher-order functions; and parametrization by shape can be achieved with higher-kinded types (and can also be encoded with abstract types).

```
newtype Fix f a = In{out :: f a (Fix f a)}
class BiFunctor f where
  bimap :: (a → b) → (c → d) → f a c → f b d
  fmap2 :: (c → d) → f a c → f a d
  fmap2 = bimap id
map :: BiFunctor f ⇒ (a → b) → Fix f a → Fix f b
map f = In ∘ bimap f (map f) ∘ out
cata :: BiFunctor f ⇒ (f a r → r) → Fix f a → r
cata f = f ∘ fmap2 (cata f) ∘ out
ana :: BiFunctor f ⇒ (r → f a r) → r → Fix f a
ana f = In ∘ fmap2 (ana f) ∘ f
hylo :: BiFunctor f ⇒ (a → f c a) → (f c b → b) → a → b
hylo f g = g ∘ fmap2 (hylo f g) ∘ f
build :: (forall b . (f a b → b) → b) → Fix f a
build f = f In
```

Figure 3. Origami in Haskell

4.1 A little DGP library

Moors et al. [2006] were the first to point out that Scala is expressive enough to be a DGP language; they showed how to encode the origami patterns (analogues of the COMPOSITE, ITERATOR, VISITOR and BUILDER design patterns) [Gibbons, 2006] in Scala. However, their encoding was in an object-oriented style that was somewhat heavyweight and had limitations that the original Haskell version did not have. We feel that this object-oriented style, while perhaps more familiar to an object-oriented programmer as Moors et al. intended, does not show the full potential of Scala from a generic programmer's perspective. We present an alternative encoding of the origami patterns that is essentially a direct translation of the Haskell solution and has the same extensibility properties.

Figure 3 shows the Haskell implementation of the origami patterns. Figure 4 shows the translation of this Haskell code into Scala. The key idea is to encode type classes through implicit parameters (see Section 3.5) rather than using the object-oriented style proposed by Moors et al.. The newtype *Fix* is mapped into a trait, and its constructor *In* into a case class; the type class *BiFunctor* maps into a trait; and the origami operations map into Scala definitions with essentially the same signatures. (In Scala, implicit parameters can only occur in the last parameter position.)

There are three things to note in the Scala version. Firstly, because evaluation in Scala is strict, we cannot just write the following in the definition of *cata*:

```
f ∘ ft.fmap2 (cata[a, r, F] (f)) ∘ (_.out)
```

(the syntax (_.*m*) is syntactic sugar for (*x* ⇒ *x.m*); in other words, '_' denotes an 'anonymous' lambda variable). Under strict evaluation, this would expand indefinitely; we have to write it more awkwardly using application rather than composition.

Secondly, in Scala there are no higher-ranked types. However, we can encode them by wrapping methods in objects. Furthermore, because Scala supports structural subtyping, we can define anonymous classes. The first argument *f* of *build* is typed with an anonymous class with a (type-parametrized) *apply* method, which encapsulates the rank-2 type.

Thirdly, we introduced both the trait *Fix* and the case class *In*, to make the correspondence with the Haskell code clearer. However, it would be more within the Scala idiom to combine these two into one (dropping one of the two names):

```
case class Fix[F[_,_], a] (out : F[a, Fix[F, a]])
```

```
trait Fix[F[_,_],a]
case class In[F[_,_],a](out:F[a,Fix[F,a]]) extends Fix[F[_,_],a]
trait BiFunctor[F[_,_]] {
  def bimap[a,b,c,d]:(a⇒b)⇒(c⇒d)⇒F[a,c]⇒F[b,d]
  def fmap2[a,c,d]   :(c⇒d)⇒F[a,c]⇒F[a,d] = bimap(id)
}
def map[a,b,F[_,_]](f:a⇒b)(t:Fix[F,a])(implicit ft:BiFunctor[F]):Fix[F,b] =
  In[F,b](ft.bimap(f)(map[a,b,F](f))(t.out))
def cata[a,r,F[_,_]](f:F[a,r]⇒r)(t:Fix[F,a])(implicit ft:BiFunctor[F]):r =
  f(ft.fmap2(cata[a,r,F](f))(t.out))
def ana[a,r,F[_,_]](f:r⇒F[a,r])(x:r)(implicit ft:BiFunctor[F]):Fix[F,a] =
  In[F,a](ft.fmap2(ana[a,r,F](f))(f(x)))
def hylo[a,b,c,F[_,_]](f:a⇒F[c,a])(g:F[c,b]⇒b)(x:a)(implicit ft:BiFunctor[F]):b =
  g(ft.fmap2(hylo[a,b,c,F](f)(g))(f(x)))
def build[a,F[_,_]](f:{def apply[b]:(F[a,b]⇒b)⇒b}) = f.apply(In[F,a])
```

Figure 4. Origami in Scala

```
trait ListF[a,r]
case class Nil[a,r] extends ListF[a,r]
case class Cons[a,r](x:a,xs:r) extends ListF[a,r]
implicit object biList extends BiFunctor[ListF] {
  def bimap[a,b,c,d] = f⇒g⇒ {
    case Nil()       ⇒Nil()
    case Cons(x,xs)⇒Cons(f(x),g(xs))
  }
}
type List[a] = Fix[ListF,a]
def nil[a]:List[a] = In[ListF,a](Nil())
def cons[a] = (x:a)⇒(xs:List[a])⇒In[ListF,a](Cons(x,xs))
```

Figure 5. Lists as a fixpoint

4.2 Using the library

Figure 5 captures the shape of lists in a type constructor *ListF*; the two possible shapes for lists are defined with the case classes *Nil* and *Cons*. The *BiFunctor* object defines the *bimap* operation for the list shape. Lists are obtained simply by applying *Fix* to *ListF*. The figure also shows functions *nil* and *cons* that play the role of the two constructors for lists.

We can now define operations on lists using the origami operators. A simple example is the function that sums all the elements of a list of integers:

```
def sumList = cata[Int,Int,ListF] {
  case Nil()       ⇒0
  case Cons(x,n)⇒x+n
}
```

4.3 Evaluation of the approach

Figure 6 presents Moors et al.'s object-oriented encoding of the origami operators (slightly adapted due to intervening changes in Scala syntax), and Figure 7 shows the specialization to lists. Compared to this object-oriented (OO) encoding, our more functional (FP) style has some advantages. The most significant difference between the two is that the OO encoding favours representing operations as methods attached to objects, and provided with a distinguished 'self' parameter, whereas the FP encoding favours representing operations as global functions, independent of any object. In particular, in the OO encoding of the type class *BiFunctor*, the method *bimap* takes just two functions, whereas in the FP encoding it takes a data structure too; the OO encoding of the *cata* operation is as a method of the class *In*, with a recursive data structure as a 'self' parameter, whereas the FP encoding is as a global function, with the recursive data structure passed explicitly. The OO approach requires more advanced language features, and leads to problems with extensibility, as we shall discuss.

```
trait ListF extends BiFunctor[ListF]
case class NilF[a,b] extends ListF {
  type A=a; type B=b
  def bimap[c,d](f:a⇒c,g:b⇒d):NilF[c,d] = NilF()
}
case class ConsF[a,b](x:a,xs:b) extends ListF {
  type A=a; type B=b
  def bimap[c,d](f:a⇒c,g:b⇒d):ConsF[c,d] =
    ConsF(f(x),g(xs))
}
type List[A] = Fix[ListF,A]
```

Figure 7. Lists as a fixpoint, after Moors et al. [2006]

The dependence on the self parameter in the OO encoding requires some sophisticated type machinery: Scala's *explicit self types*. This is seen in the definition of the trait *BiFunctor*:

```
trait BiFunctor[S<:BiFunctor[S]] ... {self:S⇒...}
trait ListF extends BiFunctor[ListF]
```

Note that *ListF* is given a recursive type bound, and that the *S* parameter of *BiFunctor* is given both an upper bound (namely *BiFunctor*[*S*]) and a lower bound (through the **self** clause, explicitly specifying the self type: an 'instance of the type class' such as *ListF* cannot instantiate the *S* parameter to anything more specific than *ListF* itself). Moors et al. [2006] explain the necessity of this construction for guaranteeing type safety; it is not required at all in the FP encoding.

A second characteristic of the OO encoding is the attachment of operations to objects as methods; for example, *cata* is a method

```
trait TC {type A; type B}
trait BiFunctor[S<:BiFunctor[S]] extends TC {
  self: S ⇒
  def bimap[c,d] (f: A ⇒ c, g: B ⇒ d): S {type A = c; type B = d}
}
trait Fix[S<:TC, a] {
  def map[b] (f: a ⇒ b): Fix[S,b]
  def cata[b] (f: S {type A = a; type B = b} ⇒ b): b
}
case class In[S<:BiFunctor[S],a] (out: S {type A = a; type B = Fix[S,a]}) extends Fix[S,a] {
  def map[b] (f: a ⇒ b): Fix[S,b] = In (out.bimap (f, _.map (f)))
  def cata[b] (f: S {type A = a; type B = b} ⇒ b): b = f (out.bimap (id, _.cata (f)))
}
def ana[s<:BiFunctor[s],a,b] (f: b ⇒ s {type A = a; type B = b}) (x: b): Fix[s,a] =
  In (f (x).bimap (id, ana (f)))
def hylo[s<:BiFunctor[s],a,b,c] (f: b ⇒ s {type A = a; type B = b}, g: s {type A = a; type B = c} ⇒ c) (x: b): c =
  g (f (x).bimap (id, hylo[s,a,b,c] (f,g)))
trait Builder[S<:BiFunctor[S],a] {
  final def build (): Fix[S,a] = bf (In[S,a])
  def bf[b] (f: S {type A = a; type B = b} ⇒ b): b
}
```

Figure 6. Origami in Scala, after Moors et al. [2006]

of the case class *In*, rather than a global function. This works smoothly for operations consuming a single distinguished instance of the recursive datatype, such as *cata*. However, it doesn't work for operations that produce rather than consume, and take no instance, such as *ana*; these appear outside the case class instead. (And of course, it is well-known [Bruce et al., 1995] that it doesn't work well for binary methods such as 'zip' either.)

In addition to the awkward asymmetry introduced between *cata* and *ana*, the association of consumer methods with a class introduces an extensibility problem: adding new consumers, such as monadic map [Meijer and Jeuring, 1995], paramorphism [Meertens, 1992], or idiomatic traversal [Gibbons and Oliveira, 2008], requires modifications to existing code. Moors et al. [2006] address this second problem through an 'extensible encoding', expressed in terms of *virtual classes* — that is, nested classes in a superclass that are overridable in a subclass. Since Scala does not provide such a construct, this virtual class encoding has itself to be encoded in terms of type members of the enclosing class (which are overridable). No such sophistication is needed in the FP approach: a new origami operator is a completely separate function.

Restricting attention now to the FP approach we describe, how does the Scala implementation compare with the Haskell one? Scala is rather more noisy than Haskell, for a variety of reasons: the use of parentheses rather than simple juxtaposition for function application; additional type annotations, for example in indicating that *bimap* is parametrized by the four types a, b, c, d; the lack of eta reduction because of strictness, as discussed above. However, the extra noise is not too distracting — and indeed, the extra explicitness in precedence might make this kind of higher-order datatype-generic programming more accessible to those not fluent in Haskell conventions.

On the positive side, the translation is quite direct, and the encoding rather transparent; the code in Figure 4 is not that much more intimidating than that in Figure 3. Scala even has some lessons to teach Haskell; for example, the 'anonymous case analysis', as used in the definitions of *biList* and *sumList*, would be nice syntactic sugar for the Haskell idiom '$\lambda x \to$ **case** x **of** $\ldots$'.

To summarize, one sometimes gets the impression from the literature that one has to accept Haskell before one can start contemplating datatype-generic programming. We believe that this need not be the case, and indeed that Scala is also a reasonable datatype-generic programming language.

5. Generic Programming with Open Datatypes

As we discussed in Section 3.2, Scala readily supports a form of algebraic datatypes via case classes. It turns out that these algebraic datatypes are quite expressive, being effectively equivalent to Haskell's *generalized algebraic datatypes* (GADTs) [Peyton Jones et al., 2006]. However, unlike the algebraic datatypes found in most functional programming languages, Scala allows the easy addition of new variants to a datatype. In this section, we see how to exploit this as a basis for a generic programming library with open representations and hence support for *ad-hoc* cases.

5.1 Type representations and generic functions

The trait *Rep*[*A*] in Figure 8 is a datatype of type representations. The three objects *RUnit*, *RInt*, *RChar* are used to represent the basic types *Unit*, *Int* and *Char*; these objects can be implicitly passed to functions that accept implicit values of type *Rep*[*A*]. The case classes *RPlus* and *RProd* handle sums and products, and the *RView* case class can be used to map datatypes into sums of products (and vice versa). The first argument of *RView* should correspond to an isomorphism, which is defined as:

```
trait Iso[A,B] {   // from and to are inverses
  def from: A ⇒ B
  def to  : B ⇒ A
}
```

For example, the isomorphism between lists and their sum of products representation is given by *listIso*:

```
trait Rep[A]
implicit object RUnit  extends Rep[Unit]
implicit object RInt   extends Rep[Int]
implicit object RChar  extends Rep[Char]
case class RProd[A,B] (ra:Rep[A], rb:Rep[B])
    extends Rep[(A,B)]
case class RPlus[A,B] (ra:Rep[A], rb:Rep[B])
    extends Rep[Either[A,B]]
case class RView[A,B] (iso:Iso[B,A], r:() ⇒ Rep[A])
    extends Rep[B]
implicit def RepProd[a,b]
  (implicit ra:Rep[a], rb:Rep[b]) = RProd(ra,rb)
implicit def RepPlus[a,b]
  (implicit ra:Rep[a], rb:Rep[b]) = RPlus(ra,rb)
```

Figure 8. Type representations in Scala.

```
def fromList[a] = (l:List[a]) ⇒ l match {
  case Nil      ⇒ Left( {})
  case (x::xs) ⇒ Right(x,xs)
}
def toList[a] = (s:Either[Unit,(a,List[a])]) ⇒ s match {
  case Left(_)        ⇒ Nil
  case Right((x,xs)) ⇒ x::xs
}
def listIso[a] =
  Iso[List[a],Either[Unit,(a,List[a])]](fromList)(toList)
```

Note that the second argument of *RView* should be lazily constructed. Unfortunately, Scala forbids the declaration of by-name arguments in case classes, so we have to encode call-by-name manually using the conventional 'thunk' technique.

As a simple example of a generic function, we present a serializer. The idea is that, given some *representable* type *t*, we can define a generic binary serializer by case analysis on the structure of the representation of *t*:

```
def ser[t] (x:t) (implicit r:Rep[t]):String =
  r match {
    case RUnit        ⇒ ""
    case RInt         ⇒ encodeInt(x)
    case RChar        ⇒ encodeChar(x)
    case RPlus(a,b)  ⇒ x.fold("0"+ser(_)(a),
                              "1"+ser(_)(b))
    case RProd(a,b) ⇒ ser(x._1)(a)+ser(x._2)(b)
    case RView(i,a)  ⇒ ser(i.from(x))(a())
  }
```

For the purposes of presentation we encode the binary representation as a string of 0's and 1's rather than a true binary stream. The arguments of *ser* are the value *x* of type *t* to encode and a representation of *t* (that may be passed implicitly). For the *Unit* case, we just return an empty string; for *Int* and *Char* we assume the existence of primitive encoders *encodeInt* and *encodeChar*. The case for sums applies the *fold* method (defined in the *Either* trait) to the value *x*; in case *x* is an instance of *Left* we encode the rest of the value and prepend 0; in case *x* is an instance of *Right* we encode the rest of the value and prepend 1. The case for products concatenates the results of encoding the two components of the pair. Finally, for the view case, we convert the value *x* into a sum of products type and apply the serialization function to that.

5.2 Open type representations and ad-hoc cases

In Scala, datatypes need not necessarily be closed to extension. This means that it is possible to introduce new variants; in the case of type representations, it means that we can add new constructors for type representations. This is useful for ad-hoc cases in generic functions — that is, to provide a behaviour different from the generic one for a particular datatype in a particular generic function.

For example, suppose that we want to use a different encoding of lists than the one derived generically: it suffices to encode its length, followed by the encodings of each of the list elements. For long lists, this encoding is likely to be more efficient than the generic behaviour obtained from the sum of products view, which essentially encodes the length in unary rather than binary format. In order to be able to define an ad-hoc case, we first need to extend our type representations with a new case for lists.

```
case class RList[A] (a:Rep[A]) extends
  RView[Either[Unit,(A,List[A])],List[A]]
    (listIso, () ⇒ RPlus(RUnit,RProd(a,RList(a))))
implicit def RepList[a] (implicit a:Rep[a]) =
  RList(a)
```

This is achieved by creating a subtype of *RView*, using the isomorphism between lists and their sum of products representation. Notice that *RList* depends on itself; had we not made this representation parameter lazy, the representation would unfold infinitely, causing a stack overflow. The function *RepList* yields a default implicit representation for lists, given a representation of the elements.

With the extra case for lists we could have an alternative serialization function with a special case for lists:

```
def ser1[t] (x:t) (implicit r:Rep[t]):String =
  r match {
    ...  // just like ser
    case RList(a)    ⇒ ser1(x.length)+
      x.map(ser1(_)(a)).foldRight("")((x,y) ⇒ x+y)
    case RView(i,a) ⇒ ser1(i.from(x))(a())
  }
```

The definition of ser_1 is essentially the same as *ser* except that there is an extra case for lists. As we explained before, this special case for lists encodes the length of the list followed by the encodings of its elements.

5.3 Evaluation of the approach

The Scala approach that we have proposed in this section compares favourably with the Haskell approach using GADTs to encode type representations. While it is true that the code to define the representation type is somewhat more verbose than the Haskell equivalent, we no longer need to create a separate type class to allow implicit construction of representations. Implicit representations may not be necessary for a generic programming library, but they are very convenient, and nearly all approaches provide them. The definition of generic functions using type representations is basically as easy in Scala as in Haskell; no significant additional verbosity is required.

In Haskell, it is very hard to extend a datatype with new variants. In contrast, in Scala, adding a new variant is essentially the same as adding a new subclass. From a generic programming point of view, the lack of extensible representations is awkward, because it prevents the design of modular generic programming libraries. This has been realised by many researchers in the past [Hinze and Peyton Jones, 2000, Lämmel and Peyton Jones, 2005, Hinze, 2006, Oliveira et al., 2006]. More recently, a number of clever approaches using Haskell type classes has been used to overcome the problem of allowing extensible representations. However, those approaches are hard to grasp, and not as direct as one could wish for. A different approach, extending Haskell with open datatypes

```
trait Generic[G[_]] {
  def unit       : G[Unit]
  def int        : G[Int]
  def char       : G[Char]
  def plus[a,b]  : G[a] ⇒ G[b] ⇒ G[Either[a,b]]
  def prod[a,b]  : G[a] ⇒ G[b] ⇒ G[(a,b)]
  def view[a,b]  : Iso[b,a] ⇒ (⇒ G[a]) ⇒ G[b]
}
```

Figure 9. The trait *Generic*.

and open functions, has been proposed by Löh and Hinze [2006], but so far that extension is not supported by any compiler. With Scala, new cases for type representations can be modularly and *easily* added, and we can define generic functions with ad-hoc cases.

There is an extra advantage of Scala's case classes when compared to Löh and Hinze's open datatypes. In Scala, we can mark a trait as **sealed**, which prohibits direct subclassing of that trait outside the module defining it. Still, we can extend subclasses even in a different module. Therefore, we could have marked the trait *Rep*[*A*] as sealed; modular extension of *RView* would still be allowed. The nice thing about this solution is that we can be sure that a fixed set of patterns is exhaustive, so it is easier to avoid pattern matching errors. Scala even has coverage checking of patterns when using case analysis on values of sealed types, warning of any missing cases.

6. Generic Programming with Type Classes

In Section 3.5 we saw how to encode type classes with implicit parameters. This section builds on that encoding, showing how generic programming techniques based on type classes can be defined in Scala. In particular, we will look at the "Generics for the Masses" (GM) technique by Hinze [2006] (Moors [2007] provides an alternative Scala tutorial on this technique), and discuss two distinct mechanisms in Scala for reusing generic functions: *reuse by inheritance* and *local redefinition*.

6.1 Generics for the masses, in Scala

Hinze's GM technique allows the definition of generic functions within Haskell 98. A generic function can be encoded as an instance of a type class *Generic*. Another type class (the *Rep* class) defines a function *rep* that can be used to construct type representations automatically.

Figure 9 presents a translation of one of the variants of Hinze's *Generic* class. Note that we use Scala's support for higher kinds to parametrize the trait *Generic* with a type-constructor *G*. The idea is that instances of the trait *Generic* represent generic functions over sums of products (much like the approach presented in Section 5). A generic function is defined by giving cases for sums, products, the unit type and also a few built-in types such as *Int* and *Char*. For sums and products, which have type parameters, we need extra arguments that define the generic functions for values of those type parameters. The *view* case uses isomorphisms to adapt generic functions to existing datatypes; once again, the '⇒' before the type *G*[*a*] signals that that parameter is passed by name.

For example, suppose that we want a generic function to count the number of values contained in data structures. Such functions have type $A \Rightarrow Int$ for various types A of data structure; we therefore introduce a parametrized signature of this type, as a case class:

```
case class Size[A] (size : A ⇒ Int)
```

Thus, a value of type *Size*[*A*] is basically a function of type $A \Rightarrow Int$ — that is, it is a record with a single field *size* of that type.

```
trait MySize extends Generic[Size] {
  def unit       = Size (x ⇒ 0)
  def int        = Size (x ⇒ 0)
  def char       = Size (x ⇒ 0)
  def plus[a,b]  = a ⇒ b ⇒ Size (_.fold (a.size, b.size))
  def prod[a,b]  = a ⇒ b ⇒ Size (x ⇒
    a.size (x._1) + b.size (x._2))
  def view[a,b]  = iso ⇒ a ⇒ Size (x ⇒
    a.size (iso.from (x)))
}
```

Figure 10. A generic function for counting values.

(Note that we have combined the trait and the case class into one definition here, as discussed in Section 4.1.)

Now the definition of counting has to specify the appropriate behaviour for units, sum, products, and so on. This is done by defining a concrete subtype of the trait *Generic*[*Size*], giving concrete implementations of each of the methods, as in Figure 10. Each case is a value of type *Size*[*A*] for some *A*, which is obtained by applying the constructor *Size* to a function of type $A \Rightarrow Int$. For flexibility, we count zero for each of the three base cases; this will be overridden later. For sums, products and user-defined datatypes, we do the obvious thing: choosing the appropriate branch of a sum, adding the counts of the two components of a product, and unpacking a view and recursively counting its contents.

6.2 Constructing type representations

A generic function is encoded as a value of type *Generic*[*G*] for some *G* (such as $G = Size$, above); in order to decode this value, we make use of another type representation. Figure 11 presents a representation *Rep*[*T*] of types *T*. This is a trait with a single method *accept*, which takes an encoded generic function of type *Generic*[*g*]; it decodes the generic function to extract the specific case at type *T*, which will be of type *g*[*T*]. Again, the Scala implementation is almost a transliteration of the Haskell type class version, except that it uses implicit parameters instead of type classes.

We will now show how to define representations of user-defined datatypes, using Scala's lists as an illustration. Essentially, for each datatype *T* we want to represent, we need to create a value of type *Rep*[*T*]. In the case of lists, such a value can be defined as follows:

```
def listRep[a,g[_]] (a : g[a])
  (implicit gen : Generic[g]) : g[List[a]] = {
    import gen._
    view (listIso[a]) (plus (unit) (prod (a)
      (listRep[a,g] (a) (gen))))
  }
implicit def RList[a] (implicit a : Rep[a]) =
  new Rep[List[a]] {
    def accept[g[_]] (implicit gen : Generic[g]) =
      listRep[a,g] (a.accept[g] (gen)) (gen)
  }
```

(The **import** declaration allows unqualified use of the methods *view*, *plus*, and so on of the object *gen*.) Here, we use the auxiliary *listRep* definition to construct the right *Generic* value following the sum-of-product structure. Note that *listIso* is the isomorphism presented in Section 5.2. Using *listRep*, we can then easily define the representation *RList* for our lists.

```
trait Rep[T] {
  def accept[g[_]] (implicit gen: Generic[g]): g[T]
}
implicit def RUnit = new Rep[Unit] {
  def accept[g[_]] (implicit gen: Generic[g]) =
    gen.unit
}
implicit def RInt = new Rep[Int] {
  def accept[g[_]] (implicit gen: Generic[g]) =
    gen.int
}
implicit def RChar = new Rep[Char] {
  def accept[g[_]] (implicit gen: Generic[g]) =
    gen.char
}
implicit def RPlus[a,b] (implicit a: Rep[a], b: Rep[b]) =
  new Rep[Either[a,b]] {
    def accept[g[_]] (implicit gen: Generic[g]) =
      gen.plus (a.accept[g] (gen)) (b.accept[g] (gen))
}
implicit def RProd[a,b] (implicit a: Rep[a], b: Rep[b]) =
  new Rep[(a,b)] {
    def accept[g[_]] (implicit gen: Generic[g]) =
      gen.prod (a.accept[g] (gen)) (b.accept[g] (gen))
}
```

Figure 11. Representations for generic functions.

6.3 Applying generic functions

We can now define a method *size* that provides an easy-to-use interface for the generic function encoded by *Size*: this takes a value of a representable type *a* and returns the number of elements counted.

```
def size[a] (x: a) (implicit rep: Rep[a]) =
  rep.accept[Size].size (x)
```

We defined *MySize* as a trait instead of an object so that it can be extended, as we discuss in more detail in Sections 6.4 and 6.5. We may, however, be interested in having an object that simply inherits the basic functionality defined in *MySize*. Furthermore, this object can be made implicit, so that methods like *rep* can automatically infer this instance of *Generic*.

```
implicit object mySize extends MySize
```

Of course, this will return a count of zero for any data structure; we show next how to override it with interesting behaviour.

6.4 Reuse via inheritance

The trait *MySize* defines a template for functions that counts values in a data structure; however, if no functionality is overridden, the resulting generic function always returns zero. The trait *MySize* becomes more useful when some of its functionality is overridden. In object-oriented languages like Scala we can use inheritance for defining new generic functions, by overriding functionality of other generic functions. In languages without inheritance (like Haskell), this kind of reuse is more difficult to achieve. For example, suppose that we wanted to define a generic function that counts the integers in some structure. Using inheritance, all we have to do is to extend *MySize* and override the case for integers so that it counts 1 for each integer value.

```
trait CountInt extends MySize {
  override def int = Size (x ⇒ 1)
}
def countInt[a] (x: a) (implicit rep: Rep[a]) =
  rep.accept[Size] (new CountInt {}).size (x)
```

With the help of *CountInt* we can define a method *countInt* to count the integers in some structure of representable type. The ability to explicitly pass an alternative 'dictionary' is essential to the definition of the method *count*, since we need to parametrize the *accept* method with an instance of *Size* other than the implicitly inferred one.

Using such generic functions is straightforward. The following snippet defines a list of integers *test* and applies *countInt* to this list.

```
val test = List (3,4,5)
def countTest = countInt (test)
```

Note that the **implicit** parameter for the type representations is not needed, because it can be inferred by the compiler (since we provided an **implicit object** *RList*).

6.5 Local redefinition

Suppose that we want to count the instances of the type parameter in an instance of a parametric datatype such as lists. It is not possible to specialize *Generic* to define such a function directly, because there is no way of distinguishing values of the type parameter from other values that happen to be stored in the structure. For example, we could have a parametric binary tree that has an auxiliary integer at each node that is used to store the depth of the tree at that node (this could be useful to keep the tree balanced). If the elements of the tree are themselves integers, we cannot count them without also counting the balance information.

```
val testTree = Fork (2, Fork (1, Value (6), Value (1)), Value (5))
val five = countInt (testTree)    // returns 5
```

To solve this problem we need to account for the representations of the type parameters of a parametric type. The method *listRep*, for example, needs to receive as an argument a representation of type $g[a]$ for its type parameter. A similar thing happens with our binary trees. Assuming that the equivalent method is called *btreeRep*, we can provide a special-purpose counter for our trees that counts only the values of the type parameter.

```
def countOne[a] = Size ((x: a) ⇒ 1)
def countTree[a] (x: Tree[a]) =
  btreeRep[a, Size] (countOne[a]).size (x)
val three = countTree (testTree)    // returns 3
```

The idea here is to replace the default behaviour that would be used for the type parameter (as inferred from the type) by user-defined behaviour specified by *countOne*.

6.6 Evaluation of the approach

Like the two previous generic programming approaches, the GM technique can be more verbose in Scala than in Haskell. For example, in the definitions of instances of the trait *Rep* (like *RUnit*, *RChar* or *RProd*), we need to state the implicit argument of the *accept* method and the type constructor argument *g* for each instance; this is not necessary in the Haskell version.

In terms of functionality, the Scala solution can provide all the functionality present in the Haskell solution, including the ability to handle local redefinitions, and more. We can easily reuse one generic function to define another through inheritance, as demonstrated in Section 6.4; with the Haskell approaches, this kind of reuse is harder to achieve. The only mechanism that we know of that comes close, in terms of simplicity, to this form of reuse is Generic Haskell's *default cases* [Löh, 2004].

Another nice aspect that the Scala approach reveals is the ability to override an implicit parameter. The *accept* method of *Rep* takes an implicit argument of type *Generic*[*g*]. When we defined the generic *countInt* function (see Section 6.4), we needed to override that argument. This was easily achieved in Scala just by explicitly passing an argument; it would be non-trivial to achieve the same effect in Haskell using type classes, since dictionaries are *always* implicitly passed. Note that we also explicitly override an implicit parameter in the definition of *countTree* (since the first argument of *btreeRep* is implicit by default).

Finally, it is interesting to observe that, when interpreted in an object-oriented language, the GM approach essentially corresponds to the VISITOR pattern. While this fact is not entirely surprising — the inspiration for GM comes from encodings of datatypes; and encodings of datatypes are known to be related to visitors [Buchlovsky and Thielecke, 2005, Oliveira, 2007] — it has not been observed in the literature before. As a consequence, many of the variations observed by Hinze have direct correspondents in variations of visitors and we may hope that ideas developed in the past in the context of visitors may reveal themselves to be useful in the context of generic programming. Oliveira [2007] explored this and has shown, for example, both how solutions to the expression problem [Wadler, 1998] using visitors can be adapted to GM and how solutions to the problem of extensible generic functions in the GM approach can be used as solutions to the expression problem.

7. Discussion

7.1 Haskell versus Scala

Scala differs significantly from Haskell, and we were curious to know what were the advantages and disadvantages of using a language like Scala instead of Haskell for the implementation of generic programming libraries. Generally speaking, Haskell has a few (mostly minor) advantages over Scala:

Laziness: Some approaches to generic programming rely, one way or another, on laziness. While laziness comes for free in Haskell, it does not in Scala, and we need to be more conscious of evaluation order. For example, we had to adapt the origami definitions in Section 4, and introduce call-by-name arguments in the *RView* constructor in Figure 8.

Syntactic clarity: While Scala's syntax is more elegant than that of Java or C#, it is still more verbose than Haskell. In particular, we have to declare more types in Scala, and need to have extra type annotations. Also, case classes can be slightly more cumbersome than Haskell **data** declarations.

Purity: Some generic programming approaches have strong theoretical foundations that provide a good framework for reasoning. However, in a language that does not guarantee the absence of side effects, the properties that one would expect may not hold. Haskell is a purely functional programming language, which means that functions will not have *silent* side-effects; Scala provides no such guarantees.

On the other hand, Scala has its own advantages:

Open datatypes with case classes: As noted in Section 5, case classes support the easy addition of new variants to a datatype. As a consequence, we can have an extensible datatype of type representations, which allows the definition of generic functions with ad-hoc cases.

Generalized type-classes with implicit parameters: In Haskell, type class "dictionaries" are always implicitly passed to functions. However, it is sometimes convenient to explicitly construct and pass a dictionary [Kahl and Scheffczyk, 2001, Dijkstra and Swierstra, 2005]. The ability to override implicit dictionaries is a desirable feature for generic programming (see, for example, [Löh, 2004, Chapter 8]).

Inheritance: Another advantage of Scala is that we can easily reuse generic functions using inheritance. In Haskell, although we can simulate this form of reuse in several ways, we can not do it naturally.

Expressive type system: The combination of subtyping, higher-kinded types, abstract types, implicit parameters, traits and mixins (among other features) provides Scala with an impressively powerful type system. Although in this paper we do not fully exploit the expressivity, Oliveira [2007, Chapter 5] shows how Scala's type system can shine when implementing modularly extensible generic functions.

Minor conveniences: We found the support for anonymous case analysis (discussed in Section 4.3) quite neat and useful. Although it is quite rare that we need to provide type annotations in Haskell expressions, when they are needed they can be quite tricky to get right; we believe that providing type annotations in Scala is easier and more intuitive. Finally, Scala's implicits can avoid the need for some of the type classes and instances that would be needed in Haskell (see the discussion in Section 6.6).

7.2 Idiomatic Scala

Throughout this paper, we have been using a functional programming style heavily influenced by Haskell and somewhat different from conventional Scala. What are the key techniques in this programming style?

Making the most of type inference. Scala does not support type inference in the same way that Haskell does. As explained in Section 2.5, in a definition like

def *power* $(x:Int):Int = twice\,((y:Int) \Rightarrow y*y,x)$

the return type of *power* and type of the lambda-bound y can be inferred, but the type of the parameter x cannot. Although in this particular case the type annotations are not too daunting, for some definitions taking several arguments while possibly being implemented or redefined in subclasses, this can become a burden. However, a simple trick can help the compiler to (at least try to) infer argument types: use lambda expressions rather than passing parameters. That is, transform a parametrized method:

def $f\ (x_1:t_1,\ldots,x_n:t_n):t_{n+1} = e$

into a parameterless method with a higher-order value:

def $f_T : t_1 \Rightarrow \ldots \Rightarrow t_n \Rightarrow t_{n+1} = x_1 \Rightarrow \ldots \Rightarrow x_n \Rightarrow e$

Then the return type $t_1 \Rightarrow \ldots \Rightarrow t_n \Rightarrow t_{n+1}$ can (possibly) be inferred, allowing a definition without type annotations:

def $f_T = x_1 \Rightarrow \ldots \Rightarrow x_n \Rightarrow e$

We have used this transformation a few times to make the most of type inference, avoiding cluttering definitions with redundant type annotations; see for example *bimap* in Figure 4, and the methods of *Generic* in Figure 9.

Type class programming. As we have seen, type classes can be encoded with implicit parameters. However, object-oriented classes are more general than type classes, because they can contain data. It is possible to mix ideas from traditional OO programming with ideas inspired by type classes. For example, Moors et al. [2008] define the trait

```
trait Ord[T] {
  def ⩽(other: T): Boolean
}
```

in order to encode the *Ord* Haskell type class

```
class Ord t where
  (⩽) :: t → t → Bool
```

There is a significant difference between the two approaches: an instance of the trait $Ord[T]$ will contain data, since the **self** variable plays the role of the first argument; whereas an instance of the type class *Ord* is essentially a dictionary containing a binary operation, with no value of type *t*. In this paper, we use the classic Haskell type class approach instead of the more OO approach. As we saw in Section 4, sometimes merging the "type class" with the data can lead to extensibility problems that can be avoided by keeping the two concepts separate.

Encoding higher-ranked types. Some more advanced Haskell libraries exploit higher-ranked types [Odersky and Läufer, 1996]. Scala does not support higher-ranked types directly, but these can be easily encoded using a class with a single method that has some local type arguments. However, this encoding requires a new (named) class, which can significantly obscure the intent of the code. In this paper, we make use of Scala's structural types to avoid most of the clutter of the encoding. The idea is simple: the Haskell definition

$$func :: forall\ a.(forall\ b.b \to b) \to a \to a$$

would be encoded in Scala as:

$$\textbf{def}\ func[a]:\ \{\textbf{def}\ apply[b]:b \Rightarrow b\} \Rightarrow a \Rightarrow a$$

Note that the type $\{\textbf{def}\ apply[b]:b \Rightarrow b\}$ stands for *some* class with a method $apply[b]:b \Rightarrow b$. Structural types allow a definition that is nearly as short and clear as the Haskell one. The main drawback of the encoding is that we have to call the *apply* method explicitly when we want to use the higher-ranked argument; in Haskell, standard function application is used instead. As a final remark, we note that this encoding makes it very easy to use parameter bounds. For example, to enforce $b <: a$ it suffices to write

$$\textbf{def}\ func[a]:\ \{\textbf{def}\ apply[b <: a]:b \Rightarrow b\} \Rightarrow a \Rightarrow a$$

If we had used a separate named class, we would have had to parametrize that class with the extra type bound arguments [Washburn, 2008].

To our knowledge, this is the first time such an encoding for higher-ranked types as been observed in the literature. We believe that building primitive support for higher-ranked types in Scala using this encoding as a basis should be fairly simple.

7.3 Porting generic programming libraries to Scala

In Haskell there has been a recent flurry of proposals for generic programming libraries [Cheney and Hinze, 2002, Hinze, 2006, Lämmel and Peyton Jones, 2005, Oliveira et al., 2006, Hinze et al., 2006, Weirich, 2006, Hinze and Löh, 2007], all having interesting aspects but none emerging as clearly the best option. An international committee has been set up to develop a standard generic programming library in Haskell. Their first effort [Rodriguez et al., 2008] is a detailed comparison of most of the current library proposals, identifying the implementation mechanisms and the compiler extensions needed.

The majority of the features required by those libraries translate well into Scala; the three approaches investigated in this paper are quite representative of the mechanisms required by most generic programming libraries. There are however some questions about some of the Haskell features. For example, certain approaches use type class extensions such as *undecidable instances*, *overlapping instances* and *abstraction over type classes*, which rely on sophisticated instance selection algorithms implemented in the latest Haskell compilers. The details of Scala's instance selection algorithm have not been published, but we believe that it is not as powerful as those in the Haskell compilers. However, as we noted before, Scala's implicit parameters allow us to explicitly pass dictionaries, which can be used whenever automatic selection of instances fails. Therefore, in principle, translating approaches that use these features into Scala should be possible (although we may have to specify a few more details explicitly).

Something that Scala does not have is a meta-programming facility. Some of the generic programming libraries use *Template Haskell* [Sheard and Peyton Jones, 2002] to automatically generate the code necessary for type representations. In Scala those would need to be generated manually, or a code generation tool would need to be developed. The *scrap your boilerplate* approach [Lämmel and Peyton Jones, 2003] relies on the ability to automatically derive instances of *Data* and *Typeable*; in Scala there is no **deriving** mechanism, so this would entail defining some instances manually.

7.4 Where to go from here

The goal of this paper was not to promote a particular approach to generic programming. Instead, we were more generally interested in investigating how Haskell's language mechanisms used in various generic programming techniques could be adapted to Scala. The hope is that this work serves as a foundation for future work on generic programming libraries in Scala. We believe that all three approaches discussed in this paper could serve as good starting points for more complete libraries. Moreover, other approaches can still greatly benefit from the discussions and insights presented in this paper.

The approach discussed in Section 5 supports open datatypes, and consequently supports ad-hoc cases for generic functions. Still, open datatypes alone can be insufficient, and sometimes *open functions* are also required (see for example Lämmel and Peyton Jones [2005]; this would also allow the removal of the duplicated code in the function ser_1 in Section 5.2). Extending this approach with open functions is an avenue for future work. One possible way to achieve this is to use *open recursion* [Cook, 1989], which has been used, in a similar context, by Garrigue [2000].

We have not discussed the possibility of adding ad-hoc cases or open (generic) functions to the approach in Section 6. However, [Oliveira, 2007, Chapter 5] (inspired by previous work [Oliveira et al., 2006]) shows how to do so.

As we have been arguing, we found that Scala has some features that are very useful in a datatype-generic programming language. We believe that other programming languages (in particular, Haskell) can learn some lessons from Scala by borrowing these features. For example, Oliveira and Sulzmann [2008] have recently proposed a generalized class system for Haskell that is partly inspired by Scala and allows both implicit and explicit passing of dictionaries.

7.5 Acknowledgements

Some of the material from this paper is partly based on the first author's DPhil thesis [Oliveira, 2007, Chapters 2 and 4]; particular thanks are due to the DPhil examiners Ralf Hinze and Martin Odersky for their helpful advice and constructive comments. This work is supported by the EPSRC grant *Generic and Indexed Programming* (EP/E02128X).

References

K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

P. Buchlovsky and H. Thielecke. A type-theoretic reconstruction of the Visitor pattern. In *Mathematical Foundations of Programming Semantics*, 2005.

J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Haskell Workshop*, pages 90–104, 2002.

R. Cockett and T. Fukushima. About Charity. Department of Computer Science, University of Calgary, May 1992.

W. R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

A. Dijkstra and S. D. Swierstra. Making implicit parameters explicit. Technical Report UU-CS-2005-032, Department of Information and Computing Sciences, Utrecht University, 2005.

J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering, Sasaguri, Japan*, 2000.

J. Gibbons. Design patterns as higher-order datatype-generic programs. In *Workshop on Generic Programming*, pages 1–12, 2006.

J. Gibbons and B. C. d. S. Oliveira. The essence of the Iterator pattern. *Journal of Functional Programming*, 2008.

C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, Mar. 1996.

R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Principles of Programming Languages*, pages 123–137, Jan. 1994.

R. Hinze. Generics for the masses. *Journal of Functional Programming*, 16(4-5):451–483, 2006.

R. Hinze and J. Jeuring. Generic Haskell: Practice and theory. In *LNCS 2793: Summer School on Generic Programming*, Berlin, 2002.

R. Hinze and A. Löh. Generic programming, now! In *LNCS 4719: Datatype-Generic Programming*, 2007.

R. Hinze and S. Peyton Jones. Derivable type classes. In *Haskell Workshop*, 2000.

R. Hinze, A. Löh, and B. C. d. S. Oliveira. 'Scrap your Boilerplate' reloaded. In *Functional and Logic Programming*, 2006.

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

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Types in Language Design and Implementation*, pages 26–37, 2003.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: Extensible generic functions. In *International Conference on Functional Programming*, pages 204–215, 2005.

X. Leroy. Manifest types, modules, and separate compilation. In *Principles of Programming Languages*, pages 109–122, 1994.

K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing, 1996.

A. Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

A. Löh and R. Hinze. Open data types and open functions. In *Principles and Practice of Declarative Programming*, pages 133–144, 2006.

C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 2007.

L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5): 413–425, 1992.

E. Meijer and J. Jeuring. Merging monads and folds for functional programming. In *LNCS 925: Advanced Functional Programming*. Springer-Verlag, 1995.

A. Moors. Code-follows-type programming in Scala. `http://www.cs.kuleuven.be/~adriaan/?q=cft_intro`, June 2007.

A. Moors, F. Piessens, and W. Joosen. An object-oriented approach to datatype-generic programming. In *Workshop on Generic Programming*, pages 96–106, 2006.

A. Moors, F. Piessens, and M. Odersky. Generics of a higher kind. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2008.

M. Odersky. An Overview of the Scala programming language (second edition). Technical Report IC/2006/001, EPFL Lausanne, Switzerland, 2006a.

M. Odersky. Poor man's type classes. `http://lamp.epfl.ch/~odersky/talks/wg2.8-boston06.pdf`, July 2006b.

M. Odersky. Scala by example. `http://scala.epfl.ch/docu/files/ScalaIntro.pdf`, May 2007a.

M. Odersky. The Scala language specification version 2.4. `http://scala.epfl.ch/docu/files/ScalaReference.pdf`, May 2007b.

M. Odersky and K. Läufer. Putting type annotations to work. In *Symposium on Principles of programming languages*, pages 54–67, 1996.

M. Odersky and M. Zenger. Scalable component abstractions. In *Object Oriented Programming, Systems, Languages, and Applications*, pages 41–57, 2005.

B. C. d. S. Oliveira. *Genericity, Extensibility and Type-Safety in the* VISITOR *Pattern*. PhD thesis, University of Oxford, 2007.

B. C. d. S. Oliveira. Scala for Generic Programmers: Source code for the examples. `http://web.comlab.ox.ac.uk/people/Bruno.Oliveira/Scala.tgz`, June 2008.

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

B. C. d. S. Oliveira, R. Hinze, and A. Löh. Extensible and modular generics for the masses. In *Trends in Functional Programming*, pages 109–138, Apr. 2006.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *International Conference on Functional Programming*, pages 50–61, 2006.

A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. Technical Report UU-CS-2008-010, Utrecht University, 2008.

N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *LNCS 2743: European Conference on Object-Oriented Programming*, pages 248–274, July 2003.

M. Schinz. A Scala tutorial for Java programmers. `http://scala.epfl.ch/docu/files/ScalaTutorial.pdf`, May 2007.

T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Haskell Workshop*, 2002.

P. Wadler. Monads for functional programming. In *Program Design Calculi*. Springer-Verlag, 1993.

P. Wadler. The expression problem. Java Genericity Mailing list, Nov. 1998. URL `http://www.cse.ohio-state.edu/~gb/cis888.07g/java-genericity/20`.

G. Washburn. Revisiting higher-rank impredicative polymorphism in Scala. `http://existentialtype.net/2008/05/26/revisiting-higher-rank-impredicative-polymorphism-in-scala/`, May 2008.

S. Weirich. RepLib: a library for derivable type classes. In *Haskell Workshop*, pages 1–12, 2006.