JFP **20** (3 & 4): 303–352, 2010. © Cambridge University Press 2010
doi:10.1017/S0956796810000171

Scala for generic programmers

Comparing Haskell and Scala support for generic programming

BRUNO C. D. S. OLIVEIRA
ROSAEC Center, Seoul National University, 599 Gwanak-ro, Gwanak-gu, Seoul 151-744, South Korea
(*e-mail:* bruno@ropas.snu.ac.kr)

JEREMY GIBBONS
Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK
(*e-mail:* jg@comlab.ox.ac.uk)

Abstract

Datatype-generic programming (DGP) involves parametrization of programs by the shape of data, in the form of type constructors such as 'list of'. Most approaches to DGP are developed in pure functional programming languages such as Haskell. We argue that the functional object-oriented language Scala is in many ways a better choice. Not only does Scala provide equivalents of all the necessary functional programming features (such as 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). Common Haskell techniques for DGP can be conveniently replicated in Scala, whereas the extra expressivity provides some important additional benefits in terms of extensibility and reuse. We illustrate this by comparing two simple approaches in Haskell, pointing out their limitations and showing how equivalent approaches in Scala address some of these limitations. Finally, we present three case studies on how to implement in Scala real DGP approaches from the literature: Hinze's 'Generics for the Masses', Lämmel and Peyton Jones's 'Scrap your Boilerplate with Class', and Gibbons's 'Origami Programming'.

1 Introduction

Datatype-generic programming (DGP) 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 'lists of integers', whereas DGP abstracts from the 'lists of'.

There is a large and growing collection of techniques for writing datatype-generic programs. Much of the early research on DGP relied on special-purpose languages or language extensions such as Charity (Cockett & Fukushima 1992), PolyP (Jansson 2000), and Generic Haskell (Hinze & Jeuring 2002). With time, research has shifted towards more *lightweight* approaches, based on language extensions such as Scrap your Boilerplate (Lämmel & Peyton Jones 2003) and Template Haskell (Sheard & Peyton Jones 2002); more recently, DGP techniques have been encapsulated in libraries for existing general-purpose languages, such as Generics for the Masses (GM) (Hinze 2006) for Haskell and Adaptive Object-Oriented Programming (Lieberherr 1996) for C++. One key advantage of the lightweight approaches is that DGP becomes more accessible to potential users, since no new tool or compiler is required in order to enjoy its benefits. Indeed, the use of libraries or simple language extensions rather than completely new languages has greatly promoted the adoption of DGP.

Despite the rather wide variety of host languages involved in the techniques listed above, the casual observer might be forgiven for concluding, from the wealth of proposals for lightweight generic programming in Haskell (Cheney & Hinze 2002; Lämmel & Peyton Jones 2005; Hinze 2006; Hinze *et al.* 2006; Oliveira *et al.* 2006; Weirich 2006; Hinze & Löh 2007; Mitchell & Runciman 2007; Brown & Sampson 2009), that 'Haskell is the programming language of choice for discriminating datatype-generic programmers'. 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.

In particular, 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 for most familiar features cherished by datatype-generic Haskell programmers, such *parametric polymorphism*, *higher-order functions*, *higher-kinded types*, and *type-* and *constructor-classes*. (Two significant missing features are *lazy evaluation* and *higher-ranked types*.) 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* (via 'traits'). We show not only that Haskell techniques for DGP can be conveniently replicated in Scala, but also that the extra expressivity provides important additional benefits in terms of extensibility and reuse. Specifically, intricate constructions are often needed to bend the implicit dictionaries in Haskell's class system to DGP purposes – these convolutions could mostly be avoided if one had first-class dictionaries. Scala's traits mechanism provides such a facility, including the ability to pass them implicitly where appropriate.

We are not the first to consider DGP in Scala: 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 9. And of course, it is not really surprising that Scala should turn out effectively to subsume Haskell, since its design is substantially inspired by functional programming.

We are interested in finding the limitations of the general-purpose mechanisms, in order to point out their weaknesses and to promote their improvement. The aim here is not to compare particular DGP libraries, as was done by Rodriguez *et al.* (2008), but rather to consider the basic mechanisms used to implement those libraries. We claim that the language mechanisms traditionally used for implementing DGP libraries, namely *type classes* (Hall *et al.* 1996) and *generalized algebraic datatypes* (GADTs) (Peyton Jones *et al.* 2006), each have their limitations, and that it might be better to exploit a different mechanism incorporating the advantages of both. This paper explores Scala's object system as such an alternative mechanism.

We feel that our main contribution is as a call to datatype-generic programmers to look beyond Haskell, and particularly to look at Scala. Not only can Scala be used to express current approaches to DGP; in some ways – in particular, with its open datatypes, inheritance, and *implicits* mechanism – it improves upon Haskell. Some of those advantages derive from Scala's mixed-paradigm nature, and so 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. We emphasize that we are not arguing that Scala is universally superior to Haskell. Indeed, as we shall see, Scala too has some limitations. Instead, our purpose is to promote the migration of good ideas from Scala to Haskell and *vice versa*, so as to improve support for DGP in both languages.

As a secondary contribution, we show that Scala is more of a functional programming language than is typically appreciated. Scala tends to be seen primarily 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 of the origami operators, but as we show in Section 9 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 serves also as a tutorial in exploiting Scala as a multi-paradigm language.

The rest of this paper is structured as follows. Section 2 sets the scene, by reviewing two straightforward approaches to DGP in Haskell – using representation datatypes and type classes, respectively – and pointing out their limitations. Sections 3 and 4 introduce the basics of Scala, and those more advanced features of its type and class system on which we depend. Our contribution starts in Sections 5 and 6, which show how to implement in Scala the two approaches presented in Haskell in Section 2. After that, three case studies of the translation of existing DGP libraries into Scala are presented: Section 7 discusses an implementation of Hinze's GM approach (Hinze 2006); Section 8 shows a Scala implementation of *Scrap your Boilerplate with Class* (Lämmel & Peyton Jones 2005); and Section 9 presents a more functional alternative to Moors *et al.*'s encoding (2006) of the *Origami Programming* operators (Gibbons 2003, 2006) in Scala. Finally, Section 10 compares Haskell and Scala support for DGP, and briefly discusses some of the key ideas of the paper, and Section 11 concludes. Scala code for the examples is available online (Oliveira 2009b).

2 Some limitations of Haskell for datatype-generic programming

To conduct our experiments, we consider two very simple and straightforward libraries for generic programming, using representation datatypes and type classes, respectively. The purpose here is to discuss the limitations of the two mechanisms. While there are various clever tricks and workarounds for dealing with these limitations, better linguistic mechanisms would provide support more naturally, and do away with the need for these tricks in the first place. We do not intend to have a debate about whether one structural view of datatypes is better or worse than another; the issues we identify are pervasive to most generic programming libraries.

2.1 Generic programming with representation datatypes

A very natural style in which to write a generic programming library is to base it on a datatype of type representations (Cheney & Hinze 2002; Hinze *et al.* 2006; Weirich 2006), leading to a structural approach similar to that of Generic Haskell. Cheney and Hinze's *lightweight implementation of generics and dynamics* (LIGD) (Cheney & Hinze 2002) provides the earliest example of such an approach, showing how to do a kind of generic programming using only the standard Hindley–Milner type system extended with existential datatypes. The key idea is to use a parametrized datatype, with the actual parameter being (a representation of) the type index; constraints on the parameter enforce consistency between the behaviour and the type index. Since Cheney and Hinze's proposal, some Haskell implementations have been extended with GADTs, which provide additional convenience that existential datatypes alone lack; we use GADTs in this section to illustrate the approach.

Here is a representation of a family of datatypes based on sums of products:

```
data Unit     = Unit
data Sum a b  = Inl a | Inr b
data Prod a b = Prod a b

data Rep t where
  RUnit  :: Rep Unit
  RInt   :: Rep Int
  RChar  :: Rep Char
  RSum   :: Rep a → Rep b → Rep (Sum a b)
  RProd  :: Rep a → Rep b → Rep (Prod a b)
```

The types *Unit*, *Sum*, and *Prod* represent, respectively, the unit type and the binary sum and binary product type constructors. The datatype *Rep t* provides the structural representation of a type *t* as a sum of products built from the primitive types *Unit*, *Int*, and *Char*. For simplicity of presentation, the treatment of isomorphisms, which allows the application of generic functions to values isomorphic to a sum of products, is omitted here; for the full details, see elsewhere (Cheney & Hinze 2002; Weirich 2006).

Generic functions are defined by case analysis on the datatype of type representations. For example, here is a definition of generic equality:

```
equals :: ∀t. Rep t → t → t → Bool
equals RUnit         _ _   = True
equals RInt          t1 t2 = (t1 ≡ t2)
equals RChar         t1 t2 = (t1 ≡ t2)
equals (RSum ra rb)  t1 t2 = case (t1, t2) of
                               (Inl x, Inl y) → equals ra x y
                               (Inr x, Inr y) → equals rb x y
                               _              → False
```

```
class Total a where
  total :: a → Int

instance Total Unit where
  total = const 0

instance Total Int where
  total = id

instance (Total a, Total b) ⇒ Total (Prod a b) where
  total (Prod x y) = total x + total y

instance (Total a, Total b) ⇒ Total (Sum a b) where
  total (Inl x)  = total x
  total (Inr y)  = total y

instance Total a ⇒ Total [a] where
  total = total ∘ fromList

fromList :: [a] → Sum Unit (Prod a [a])
fromList []       = Inl Unit
fromList (x : xs) = Inr (Prod x xs)
```

Fig. 1. A generic ‘sum’ function, using type classes.

```
equals (RProd ra rb) t1 t2 = case (t1, t2) of
                                 (Prod x y, Prod x' y') → equals ra x x' ∧ equals rb y y'
```

Equality is vacuous at the unit type, represented by *RUnit*, since there is exactly one value of that type. When the type representation is *RInt*, the type constraint ensures that the two values of type t being compared are indeed integers, and so the primitive comparison on integers is used; and similarly for *RChar* and characters. For binary sums, the constructor *RSum* of the representation is applied to representations of the two summands, and these representations are used in the recursive calls; similarly for products and *RProd*.

2.2 *Generic programming with type classes*

An alternative to using datatypes for type representations is to use type classes (Lämmel & Peyton Jones 2003; Hinze 2006; Mitchell & Runciman 2007; Brown & Sampson 2009), since both techniques can be used to define *type-indexed functions*, albeit with slightly different properties (Oliveira & Gibbons 2005). Figure 1 presents a simple definition of a generic ‘sum’ function using type classes. The class *Total a* has a method *total* that takes an argument of type a and returns an integer result. There are instances for each of the sum-of-products structural types: for the unit type, zero is returned; for binary products, the results on the two components are added; for binary sums, the function is applied recursively to the appropriate case; for integers, the integer itself is returned. Instead of specially crafting an instance specific to lists, the argument is converted into sum-of-products form using the function *fromList* and then subjected to structural cases of *total*. The same structural approach can be taken for other datatypes, avoiding the need for specific definitions for those datatypes.

2.3 Limitations

The two approaches presented in Sections 2.1 and 2.2 are relatively simple to understand. Unfortunately, they are also rather simple-minded, and suffer from some limitations in terms of both convenience and expressiveness. More realistic generic programming libraries address some of these limitations, but usually at the cost of comprehensibility. We discuss these limitations next, together with some of the attempts to address them.

2.3.1 Convenience and readability

Two important considerations of convenience concerning a DGP library are how easy it is to define generic functions, and how easy it is to apply them. Defining generic functions with a datatype of type representations is usually straightforward, since all that is needed is to use pattern matching on the type representation to introduce a definition by cases. Type classes impose some overhead, since each instance declaration requires some additional code. Furthermore, the use of datatypes and pattern matching is arguably more natural than type classes and dispatching. Nonetheless, the overhead imposed by type classes is tolerable.

Using a generic function based on type representations requires that the corresponding value of the type representation is constructed. For example, to compare two pairs of integers, one needs a third argument representing the type ‘pairs of integers’:

```
testDT = equals (RProd RInt RInt) (Prod 3 4) (Prod 4 4)
```

In contrast, with the type class approach, the explicit construction of the type representation is not necessary:

```
testTC = total (Prod 3 4)
```

Not having to explicitly construct type representations is an advantage of the type class approach, and provides additional convenience over an approach based on representation datatypes.

The reality. In existing proposals for DGP libraries using datatypes of type representations, it is common to use type classes to automatically generate the values of the type representations. There are even some proposals, such as Hinze’s (GM) approach (Hinze 2006), which do not directly use a datatype of type representations, but encode one using type classes, and also use the same mechanism to generate the values of the encoded type representations. The basic idea is as follows:

```
class Representable a where
  rep :: Rep a

instance Representable Int where
  rep = RInt

instance (Representable a, Representable b) ⇒ Representable (Prod a b) where
  rep = RProd rep rep
```

(Here, an LIGD-like approach is used for illustration.) An equality function that does not need an explicit value for the type representation is definable as follows:

$$\begin{aligned}&\mathit{eq} :: \mathit{Representable}\ a \Rightarrow a \to a \to \mathit{Bool}\\&\mathit{eq} = \mathit{equals}\ \mathit{rep}\end{aligned}$$

While this design does bring some of the convenience of type classes into a datatype-based approach, the fact is that *two* different functions for equality are needed: the function *equals* defines the structural generic function, and the *eq* function provides a convenient interface to this generic function. As we shall see in Section 2.3.2, sometimes it would be handy to have functions that could take an optional argument, leaving the job of generating the value to the compiler when this argument is omitted. Having two differently-named functions for the explicit and implicit cases is awkward.

2.3.2 *Coexistence of implicit and explicit arguments*

Generic Haskell provides a simple mechanism for precisely controlling which case gets applied. For instance, using a Generic Haskell generic function $\mathit{total}\langle T\rangle$ similar to the function presented in Figure 1, it is possible to employ *local redefinition* (Löh 2004, Chapter 8) to *override* a case in a particular use of the generic function. With local redefinitions it is easy to have one variation that counts the values in a list:

$$\mathbf{let}\ \mathit{total}\langle a\rangle = \mathit{const}\ 1\ \mathbf{in}\ \mathit{total}\langle[a]\rangle[1\mathinner{..}10]$$

and another sums the (integer) values in that list:

$$\mathbf{let}\ \mathit{total}\langle \mathit{Int}\rangle = \mathit{id}\ \mathbf{in}\ \mathit{total}\langle[\mathit{Int}]\rangle[1\mathinner{..}10]$$

The *total* function in Figure 1 sums the integers in a structure. In order to provide a local redefinition to count rather than sum the elements, one might attempt to provide the following alternative instance for integers:

$$\begin{aligned}&\mathbf{instance}\ \mathit{Total}\ \mathit{Int}\ \mathbf{where}\\&\quad \mathit{total} = \mathit{const}\ 1\end{aligned}$$

However, this instance overlaps with the one already given in Figure 1. This leads to an ambiguity in instance selection; Haskell provides no mechanism for resolving the ambiguity, and disallows the coexistence of such instances. (What is needed here is some explicit mechanism for dictionaries, rather than the implicit mechanism provided by the class system. GHC's 'overlapping instances' flag does not help, because the instances are duplicated; it is useful only when one instance is more specific than the other, when it allows the compiler to select the more specific instance.)

The reality. Very few generic programming libraries in Haskell provide support for local redefinitions. In fact, we believe that currently only GM and its *extensible and modular generics for the masses* (EMGM) extension (Oliveira *et al.* 2006) provide some support for this feature, and then only partially – although there is an alternative encoding using a proposed extension to Haskell (Hinze & Löh 2009). As Hinze (2006) points out, in the GM approach, a generic counter function can be instantiated to behave like a summing function or a size function. However, this is significantly less convenient to use than the Generic Haskell solution, since two different functions are needed: one for when no local redefinitions are used, and another to explicitly pass the type representation argument with the local redefinition. Worse, first-class generic functions induce an exponential growth in the number of combinations. For example, in the case of a generic function like *everywhere* (Lämmel & Peyton Jones 2003), which in turn takes a generic function as an argument, four different variations would be needed – for each combination of allowing and disallowing local redefinitions for *everywhere* itself and for the generic function passed as an argument.

In a generic programming approach like the one shown in Figure 1, it is simply not possible to have local redefinitions without some forward planning. There have been some proposals to extend Haskell with a mechanism for choosing a particular *named instance* (Kahl & Scheffczyk 2001; Dijkstra & Swierstra 2005), but these extensions are not widely implemented. It is possible to use a simple trick to emulate named instances, as shown for example by Löh (2004); however, this still entails significant rewriting, planning ahead, and some advanced Haskell extensions.

2.3.3 Extensibility

Generic functions are useful because they work 'out of the box' for a newly introduced datatype. However, it is sometimes desirable to define a specific (non-generic) behaviour for the generic function on a particular datatype. For this to happen, the generic function needs to be *extensible*, allowing the definition of new cases for particular datatypes. From a generic programming point of view, extensible generic functions are essential for the design of modular generic programming libraries (Hinze & Peyton Jones 2000; Lämmel & Peyton Jones 2005; Hinze 2006; Oliveira *et al.* 2006). For example, abstract datatypes such as sets are often represented using a standard algebraic datatype like lists or trees, but a generic function based solely on the algebraic structure of the representation probably does not provide an appropriate implementation on the abstract type. Consider the case of equality; while structural equality is the right thing to do for most datatypes, it is wrong for an abstract datatype of sets represented as lists.

In the simple DGP approach presented in Section 2.1, it is not possible to extend the equality generic function in a modular way. In order to add a new case for sets, one must add a new constructor to the *Rep* datatype, and provide a special case for equality on sets, as follows:

```
newtype Set a = Set [a]
data Rep t where
  ...
  RSet :: Rep a → Rep (Set a)
equals :: ∀t. Rep t → t → t → Bool
...
equals (RSet a) = ...
```

On the other hand, in the type-class-based approach, the generic function can be easily extended with new cases; all that is needed is to create a new instance:

```
instance Total (Set a) where
  total (Set xs) = ...
```

In essence, approaches based on datatypes of representations usually do not support modular extensions, whereas those based on type classes do.

The reality. In recent proposals for DGP libraries, the trend is to use type classes: they can be extended more easily than datatypes, and also they make the use of generic functions quite convenient (see Section 2.1). However, even using type classes, *extensibility* can still be problematic – especially in combination with *first-class generic functions*, as we shall see in Section 2.3.4. This is the case, for example, for the original GM approach, which does not allow extensible generic functions. To address these extensibility problems, a number of clever approaches have been proposed. RepLib (Weirich 2006) uses a mix of datatypes and type classes to allow extensible generic functions; the key idea is to use a standard generic function defined on a datatype of type representations, and use that generic function as the default for another (type-class-overloaded) function that can be extended with new *ad hoc* cases. While the approach achieves its goal of supporting an extensible generic programming library, it does so at the loss of some usability and understandability: it relies on several non-standard extensions, and it requires the programmer to write generic functions in two different styles, namely using datatypes and type classes. Ultimately, a programmer needs to understand quite a bit of the mechanics of the generic programming library and some advanced Haskell features to use the approach effectively. The EMGM approach (Oliveira *et al.* 2006) addresses the extensibility limitations of the original GM proposal requiring only a common extension to Haskell 98, namely, multiple-parameter type classes; however, writing generic functions in EMGM (and in the original GM) is not as direct as using a datatype of type representations. The original *Scrap your Boilerplate* (SyB) approach (Lämmel & Peyton Jones 2003) approach does not support extensible generic functions; to address this problem, an alternative implementation (Lämmel & Peyton Jones 2005) of SyB using type classes has been proposed, but this approach uses many non-standard extensions and tricks.

A different solution (Löh & Hinze 2006) to the extensibility problem consists of extending Haskell with open datatypes and open functions. This would have some important advantages, especially from a usability point of view, since the natural style of writing a generic function using pattern matching on the type representation would be preserved. However, to date that extension is not supported by any compiler.

2.3.4 *First-class generic functions and generic function abstraction*

The SyB approach has shown the utility of first-class generic functions for generic traversals and queries. With a datatype of type representations, it is straightforward to write such functions (Hinze 2003). For example, considerthe function *everywhere*:

$$everywhere :: (\forall b.\ Rep\ b \to b \to b) \to Rep\ a \to a \to a$$

It takes as an argument a generic function that transforms a value of type b into another value of the same type; it also takes a representation of some type a and a value of that type, and returns a value of the same type. In this approach, first-class generic functions are quite simple and natural. However, with the simple type-class approach, it is far from obvious how to write the type of *everywhere*. The key problem is that *everywhere* needs to be applicable to *any* generic function as its first argument. In pseudo-code, the intended type is as follows:

$$everywhere :: Everywhere\ a \Rightarrow (\forall b.\ g\ b \Rightarrow b \to b) \to a \to a$$

That is, *everywhere* should take a generic function defined in an arbitrary type class g as the first argument. However, Haskell does not support type class abstraction, and the type class constraint $g\ b \Rightarrow \ldots$ is not valid.

In summary, while a datatype-based approach trivially supports first-class generic functions, a type-class-based approach stumbles over the fact that type classes cannot be abstracted.

The reality. There are some generic programming approaches based on type classes that do not have a problem with first-class generic functions. Interestingly enough, these show a strong correlation with the approaches that have a problem with extensibility. In other words, there seems to be a conspicuous relationship between extensibility and first-class generic functions: having one of these features makes the other feature harder to achieve (in Haskell, at least).

Using a technique proposed by Hughes (1999), it is possible to emulate type-class abstraction in Haskell using only existing extensions. This technique has been used in the ‘SyB with Class’ approach (Lämmel & Peyton Jones 2005) to allow extensible higher-order generic functions (see Section 8.1 for more details); a similar technique is used in RepLib (Weirich 2006).

2.3.5 Reuse of generic functions

Generic Haskell provides *default cases*, a mechanism that allows the reuse of generic functions (Löh 2004, Chapter 14). The motivation for this is that often minor variations of a generic function are written over and over again. For example, consider collecting variables in some datatype of abstract syntax trees. Instead of defining a function generic in the datatype but specific to the problem of collecting variables, a more general function for collecting values (Löh 2004, Chapter 9) could be reused, overriding the case for the variable type. In Generic Haskell, this idea can be expressed as follows:

```
newtype Var = V String

varcollect extends collect
varcollect⟨Var⟩(V x) = [x]
```

Neither the datatype nor the type class solutions allow for this kind of reusability; without anticipation, it is necessary to duplicate code in creating a variation of the original function.

The reality. As far as we are aware, this kind of reuse is not addressed by any generic programming library in existence, except by an early approach (Lämmel *et al.* 2000) based on Haskell records that achieves reuse between algebras by exploiting record updates. Unfortunately, although the current implementation of type classes in most Haskell compilers is based on records, the updating feature is not available for type classes.

Reuse of generic functions is akin to inheritance, and it is known how to encode inheritance in functional languages (Cook 1989). So, in theory, it should be possible to adapt existing generic programming libraries to achieve this kind of reuse via inheritance; however, any encoding introduces its own cost in terms of usability. Another alternative is to create a more general generic function that is parametrized by functions covering the different cases; but this requires anticipation, and makes the interface of the generic function more complex.

2.3.6 Exotic types

DGP techniques are applicable to some exotic types, such as datatypes with higher-kinded type arguments and nested datatypes (Hinze 2000). One example of the former is the type of generalized rose trees:

$$\textbf{data}\ GRose\ f\ a = GFork\ a\ (f\ (GRose\ f\ a))$$

The type constructor *GRose* is parametrized by a higher-kinded argument f. Datatype-based approaches to DGP comfortably support type representations for such types, and corresponding cases for generic functions:

```
data Rep t where
  ...
  RGRose :: (∀a. Rep a → Rep (f a)) → Rep a → Rep (GRose f a)
equals :: ∀t. Rep t → t → t → Bool
...
equals (RGRose f a) (GFork x xs) (GFork y ys) =
  equals a x y ∧ equals (f (RGRose f a)) xs ys
```

One small inconvenience with the datatype-based approach is that it is not possible to use nested patterns such as *RGRose RList a*: the first argument of the *RGRose* constructor is not a valid pattern, since it is not fully applied to the right number of arguments. Nevertheless, there is a workaround that allows emulation of such nested patterns (Hinze & Löh 2009).

With type-class-based approaches, support for datatypes like *GRose* does not work so smoothly. Recent versions of some Haskell compilers support recursive dictionaries in type classes, and accept the following code:

	Datatypes	*Type classes*
Convenience:		
Defining generic functions	●	◐
Using generic functions	◐	●
Implicit explicit parametrization	○[1]	○[2]
Extensibility	○	●
First-class generic functions	●	○
Reuse of generic functions	○	○
Exotic types	●	◐[3]

Fig. 2. Evaluation of the Haskell mechanisms for DGP. Key: ● = 'good', ◐ = 'sufficient', ○ = 'poor' support. Notes: (1) datatypes only allow explicit parametrization; (2) type classes only allow implicit parametrization; (3) datatypes with higher-kinded type arguments can be accommodated using undecidable instances.

```
instance (Total a, Total (f (GRose f a))) ⇒ Total (GRose f a) where
  total (GFork x xs) = total x + total xs
```

but this requires allowing undecidable type class instances. (Indeed, in older versions of some compilers, it used to be the case that such an instance would lead to non-termination of the type checker (Hinze & Peyton Jones 2000).)

A theoretically more appealing solution, suggested by Hinze and Peyton Jones (2000), would be to allow *polymorphic predicates* in the constraints. With such a feature, the following instance would be valid:

```
instance (∀a. Total a ⇒ Total (f a), Total a) ⇒ Total (GRose f a) where
  total (GFork x xs) = total x + total xs
```

The reality. As shown by the recent comparison of generic programming libraries in Haskell (Rodriguez *et al.* 2008), exotic features such as datatypes with higher-kinded type arguments and nested datatypes are not a problem for most approaches that use a datatype of type representations. However, none of the approaches based on type classes is fully capable of handling such exotic types. The limitations of type classes are to blame.

2.4 Discussion

Type classes and datatypes provide two alternative mechanisms for DGP, but neither mechanism is clearly superior to the other. Figure 2 shows the trade-offs between the two mechanisms. Datatypes provide a very natural and convenient way to define new generic functions, but they also require every value to be explicitly constructed by the programmer; this makes generic functions harder to use. Datatypes make it easy to support first-class generic functions, and they can be used to construct type representations for higher-kinded types; however, achieving extensibility is difficult. Type classes are convenient when it comes to using generic functions, since dictionaries are automatically inferred by the compiler, but they provide a somewhat less natural syntax for defining generic functions. It is easy to extend generic functions with new cases, but hard to support first-class generic functions. Exotica such as higher-kinded types and nested datatypes pose a challenge to a type-class-based implementation. Values of datatypes can only be passed explicitly, while type class dictionaries can only be passed implicitly. Neither mechanism provides an easy way to reuse generic functions.

The reality is that in Haskell it is usually possible to work around the limitations of the two mechanisms in one way or another, but doing so typically requires clever tricks or solutions that hinder usability and comprehensibility. We feel that this a symptom of inappropriate language features, and we claim that with a different mechanism, generic libraries could be defined more naturally and used more conveniently.

3 Functional programming in 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, 2007b; Schinz 2007; Odersky *et al.* 2008).

3.1 Definitions and values

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

$\textbf{def}\ square\ (x : Double) : Double = x * x$

Scala distinguishes between definitions and values. In a definition $\textbf{def}\ x = e$, the expression e will not be evaluated until the value of x is needed. Scala also offers a value definition $\textbf{val}\ x = e$, in which the right-hand side e is evaluated at the point of definition. However, only definitions can take parameters; values must be constants (although these constants can be functions).

3.2 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 a given function f twice to its argument x, we could write:

$\textbf{def}\ twice\ (f : Int \Rightarrow Int, x : Int) : Int = f\ (f\ (x))$

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

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

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

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

$$\textbf{def}\ \mathit{curryTwice}\ (f : \mathit{Int} \Rightarrow \mathit{Int})\ (x : \mathit{Int}) : \mathit{Int} = f\ (f\ (x))$$

3.3 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:

$$\textbf{def}\ \mathit{comp}\,[a,b,c]\ (f : b \Rightarrow c)\ (g : a \Rightarrow b)\ (x : a) : c = f\ (g\ (x))$$

The function *comp* is parametrically polymorphic in the three types a,b,c of the initial, intermediate and final values. Note that these type variables have to be explicitly quantified.

3.4 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, one can also pass arguments *by name*, by prefixing the type of the formal parameter 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. 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:

$$\textbf{def}\ \mathit{Then}\ (p : \mathit{Parser})\ (q : \Rightarrow \mathit{Parser}) : \mathit{Parser} = \ldots$$

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

3.5 Type inference

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

$$\textbf{def}\ \mathit{add}\ (x : \mathit{Int}) : \mathit{Unit} = \ldots$$
$$\textbf{def}\ \mathit{add}\ (x : \mathit{String}) : \mathit{Unit} = \ldots$$

This makes type inference more difficult than in languages like Haskell. Nevertheless, Scala does support a form of *local type inference* (Odersky *et al.* 2001). Thus, it is possible, most of the time, to infer the return type of a definition and the type of a lambda-bound variable. For example, one may write:

$$\textbf{def}\ \mathit{power}\ (x : \mathit{Int}) = \mathit{twice}\ (y \Rightarrow y * y, x)$$

and both the return type and the type of the lambda variable y will be inferred.

3.6 Sums, products and lists

The Scala libraries provide implementations of sums, products, and lists. For sum types, the type constructor *Either* is used. Following Haskell conventions, this type has two constructors *Left* and *Right*, injections into the sum. For example,

$$\begin{aligned}&\mathbf{val}\ \mathit{leftVal}\ \ : \mathit{Either}\,[\mathit{Int},\mathit{String}] = \mathit{Left}\ (1)\\ &\mathbf{val}\ \mathit{rightVal} : \mathit{Either}\,[\mathit{Int},\mathit{String}] = \mathit{Right}\ (\texttt{"c"})\end{aligned}$$

define two values of the type *Either* [*Int*, *String*]. One can use pattern matching to deconstruct a value of a sum type, as discussed in Section 4.2, but a more compact notation is given by the *fold* of the *Either* type. For example,

$$\mathbf{def}\ \mathit{stringVal}\ (x : \mathit{Either}\,[\mathit{Int},\mathit{String}]) = x.\mathit{fold}\ (y \Rightarrow y.\mathit{toString}\ (), y \Rightarrow y)$$

defines a function that takes a value of type *Either* [*Int*, *String*] and returns a string representing the value contained in the sum.

Products can be defined with the usual tuple notation; for example:

$$\mathbf{val}\ \mathit{prodVal} : (\mathit{Int},\mathit{Char}) = (3, \texttt{'c'})$$

To extract the components of a tuple, Scala provides methods with names consisting of an underscore followed by the component number:

$$\begin{aligned}&\mathbf{val}\ \mathit{fstVal}\ = \mathit{prodVal}.\_1\\ &\mathbf{val}\ \mathit{sndVal} = \mathit{prodVal}.\_2\end{aligned}$$

Finally, we can use the syntax *List* $(a_1, \ldots, a_n)$ to construct a list of size n with the elements a_i for $i \in [1..n]$. For example:

$$\mathbf{val}\ \mathit{list} = \mathit{List}\ (1, 2, 3)$$

builds the list with 1, 2 and 3 as elements.

4 Object-oriented programming in Scala

Scala has a rich object system, including object-oriented constructs such as concrete and abstract classes, subtyping, and inheritance familiar from mainstream languages like Java or C#. Scala also incorporates some less commonly known concepts; in particular, there is a syntactic 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*, instances of which can be decomposed using case analysis and pattern matching.

This section introduces a subset of the full Scala object system, sufficient to model all the programs in this paper.

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

This example uses traits in much the same way as one might 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*.

4.2 Objects and case classes

New object instances can be created as in most object-oriented languages, by using the **new** keyword. For example, we could define a new *Basics* object by:

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

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

```
object basics2 extends Basics
```

Scala also supports the notion of a *case class*, which simplifies the definition of functions by case analysis. In particular, case classes allow the emulation of algebraic datatypes from conventional functional languages. Figure 3 gives definitions analogous 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. The definition

```
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))
}
```

Fig. 3. Algebraic datatypes and case analysis in Scala.

of the function *ins* shows another case analysis on lists, and also demonstrates the use of *type-parameter bounds*: the list elements must be drawn from an ordered type.

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 ())))
```

4.3 Higher-kinded types

Type-constructor polymorphism and constructor classes have proven to be very useful in Haskell, allowing, among other things, the definition of concepts such as monads (Wadler 1993), applicative functors (McBride & Paterson 2008), 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* [_], a type that is itself parametrized by another type – in other words, *Container* is a type constructor. By parametrizing over the type constructor rather than a particular type *Container* [*A*], one 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, *map* would have to be homogeneous.

4.4 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

```
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)
  }
}
```

Fig. 4. An abstract datatype for sets.

hide information about internals of a component, in a way similar to their use in Standard ML (Harper & Lillibridge 1994) and OCaml (Leroy 1994). Odersky and Zenger (2005) argue that abstract types are essential for the construction of reusable components: they allow information hiding over several objects, a key ingredient of component-oriented programming.

Figure 4 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.

4.5 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), Oliveira *et al.* (2010), they can be used to emulate Haskell's type classes (Hall *et al.* 1996). Consider this approximation to the concept of a monoid (Odersky 2006a), omitting any formalization of the monoid laws:

```
trait Monoid [a] {
  def unit              : a   // unit of op
  def op (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 the binary operation being string concatenation.

```
implicit object strMonoid extends Monoid [String] {
  def unit                       = ""
  def op (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, one can now define operations that are generic in the monoid:

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

Now *reduce* can be used in the obvious way:

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

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

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

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

5 Generic programming with open datatypes

In this section, we present a Scala version of the Haskell approach based on datatypes of type representations described in Section 2.1. As shown in Section 4.2, Scala readily supports a form of algebraic datatypes, via case classes. It turns out that these algebraic datatypes are quite expressive, being effectively comparable to Haskell's GADTs. However, unlike the algebraic datatypes found in most functional programming languages, Scala allows an encoding of *open datatypes* (or, from an object-oriented programming perspective, *multi-methods* Agrawal *et al.* 1991), enabling the addition of new variants to a datatype. This section exploits this encoding as a basis for a generic programming library with *open type representations*, and hence with support for (modular) *ad hoc* cases.

```
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)
```

Fig. 5. Type representations in Scala.

5.1 *Type representations and generic functions*

The trait *Rep* [*A*] in Figure 5 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*:

```
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 by-name qualification at that argument position, 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 serial [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" + serial (_) (a), "1" + serial (_) (b))
    case RProd (a,b) ⇒ serial (x._1) (a) + serial (x._2) (b)
    case RView (i,a) ⇒ serial (i.from (x)) (a ())
  }
```

For the purposes of presentation, we encode the binary representation as a string of zeroes and ones rather than a true binary stream. The arguments of *serial* are the value x of type t to encode and a representation of t (which may be passed implicitly). For the *Unit* case, we return an empty string; for *Int* and *Char*, we assume 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 there is an instance of *Left*, we encode the rest of the value and prepend 0; in case x there 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 its sum-of-products equivalent and apply the serialization function to that.

5.2 Open type representations and ad hoc cases

In Scala, datatypes may be open to extension – that is, it is possible to introduce new variants; in the case of type representations, it means that we can add new constructors for representations of new types. 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 the length of a list, followed by the encodings of each of its elements. For long lists, this encoding is 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 this representation parameter not been made lazy, the representation would unfold indefinitely. The function *RepList* yields a default implicit representation for lists, given a representation of the elements.

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

```
def serial1 [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" + serial1 (_) (a), "1" + serial1 (_) (b))
    case RList (a)    ⇒ serial1 (x.length) +
      x.map (serial1 (_) (a)).foldRight ("") ((x, y) ⇒ x + y)
    case RView (i,a) ⇒ serial1 (i.from (x)) (a ())
  }
```

The definition of *serial*$_1$ is essentially the same as *serial*, except that there is an extra case for lists, producing an encoding of the list length followed by the encodings of its elements.

5.3 Inheritance of generic functions

The definition of the *serial*$_1$ generic function is somewhat unsatisfactory, because it involves code duplication. Scala, being an object-oriented language, supports inheritance. However, to make use of inheritance on generic functions, we need to adapt our programs to use classes instead of function definitions: the serialization function *serial* has to be rewritten as follows:

```
trait Producer [a] {
  def apply [t] (x : t) (implicit r : Rep [t]) : a
}
case class Serial extends Producer [String] {
  def apply [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" + apply (_) (a), "1" + apply (_) (b))
    case RProd (a,b)    ⇒ apply (x._1) (a) + apply (x._2) (b)
    case RView (iso,a) ⇒ apply (iso.from (x)) (a ())
  }
}
object serial extends Serial
```

The trait *Producer* defines a ‘template’ for generic producer functions such as serialization, and can be reused for other producers. The case class *Serial* is a subclass of *Producer*, implementing the *apply* method. The definition of *serial* is recovered by an object extending *Serial*. Scala treats methods named *apply* specially, allowing the use of the conventional function application notation for invocation:

we can write *o* (*arg*) instead of *o.apply* (*arg*), and *serial* (*List* (1)) instead of *serial.apply* (*List* (1)).

The advantage of writing the generic function in this style, rather than more directly using a function definition, is that inheritance allows the definition to be reused. For example, instead of repeating all the cases in $serial_1$, we could write the following:

```
case class Serial1 extends Serial {
  override def apply [t] (x : t) (implicit r : Rep [t]) : String = r match {
    case RList (a) ⇒
      apply (x.length) + x.map (apply (_) (a)).foldRight ("") ((x, y) ⇒ x + y)
    case _         ⇒ super.apply (x) (r)
  }
}
object serial1 extends Serial1
```

The case class $Serial_1$ inherits from *Serial* and adds a special case for lists, overriding the generic definition. The default case '_' uses **super** to invoke the definition of *apply* from *Serial* whenever the argument is not a list representation. The definition of $serial_1$ is once again recovered by an object extending the class $Serial_1$ that represents the generic function. The following shows how client code can use the new versions of *serial* and $serial_1$:

```
val testSerial  = serial (List (1))
val testSerial1 = serial1 (List (1))
```

5.4 Evaluation of the approach

The Scala approach presented in this section compares favourably with the Haskell approach using GADTs to encode type representations, which was presented in Section 2.1. 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 strictly 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 straightforward in Scala as in Haskell; no significant additional verbosity is involved.

In Haskell, it is difficult to extend a datatype with new variants, which has drawbacks from a generic programming point of view, as discussed in Section 2.3.3. In contrast, in Scala, adding a new variant is essentially the same as adding a new subclass. While it would be possible to overcome Haskell's extensibility limitations with the *open datatypes* proposal (Löh & Hinze 2006), Scala's case classes have an extra advantage when compared to that proposal. 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; but 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 about any missing cases.

Finally, inheritance allows one to reuse code from existing generic functions. However, generic functions need to be written using classes instead of function definitions, which is a bit more verbose and less direct. Nonetheless, reuse of generic functions is an important and useful feature to have, and the native support provided by Scala makes this feature quite usable.

6 Generic programming with type classes

This section shows how to implement a Scala equivalent for the simple generic programming approach based on type classes presented in Section 2.2. Following Section 4.5, the key idea is to use Scala's implicits mechanism instead of Haskell's type classes.

6.1 Simple extensible generic functions

Figure 6 shows a Scala implementation of the Haskell code presented in Figure 1. The trait *Total* [*A*] plays a role similar to that of the class *Total* in the Haskell version: it defines an interface containing a function *total* that takes a value of type *A* and returns an integer. The *gtotal* function provides an interface to the sum function, constructing the dictionary implicitly; this is not needed in the Haskell version. However, unlike with the type class version, such a dictionary can also be passed explicitly. The *totalUnit*, *totalInt*, *totalPair* and *totalEither* definitions are analogous to the type class instances for units, integers, products, and sums. One notable difference is the explicit selection of the dictionary in which to find a subsidiary sum function; for example, in the definition for pairs, *totalA.total* selects the sum function from the first dictionary argument. For lists, the *gtotal* function is used instead, allowing the dictionary to be automatically inferred by the compiler. (The function *fromList* was defined in Section 5.1.)

6.2 Local redefinition

With the Scala approach, local redefinition is possible. For example, we can obtain a generic function that counts the integers in a structure by using a different dictionary for integers:

```
object countInt extends Total [Int] {
  def total = x ⇒ 1
}

val x = gtotal (List (1, 2, 3))
val y = gtotal (List (1, 2, 3)) (totalList (countInt))
```

```
trait Total[A] {
  def total : A ⇒ Int
}
def gtotal[A] (x : A) (implicit t : Total[A]) : Int = t.total (x)
implicit object totalUnit extends Total[Unit] {
  def total = _ ⇒ 0
}
implicit object totalInt extends Total[Int] {
  def total = x ⇒ x
}
implicit def totalPair[a,b] (implicit totalA : Total[a], totalB : Total[b]) =
  new Total[(a,b)] {
    def total = {
      case (x,y) ⇒ totalA.total (x) + totalB.total (y)
    }
  }
implicit def totalEither[a,b] (implicit totalA : Total[a], totalB : Total[b]) =
  new Total[Either[a,b]] {
    def total = {
      case Left (x)   ⇒ totalA.total (x)
      case Right (y) ⇒ totalB.total (y)
    }
  }
implicit def totalList[a] (implicit totalA : Total[a]) = new Total[List[a]] {
  def total = x ⇒ gtotal (fromList (x))
}
```

Fig. 6. A simple generic 'sum' function.

The values *x* and *y* are computed by calling the same generic function *gtotal* on the same list; however, in the second case, a local redefinition of the dictionary to use for integers means that the result is 3 rather than 6. (Given this facility for local redefinition, it would be reasonable to make the basic function a generic 'counter', returning zero for each base case, and to expect some cases to be locally redefined for each use.)

6.3 Exotic types

In the Haskell approach using type classes, exotic types such as nested datatypes or datatypes with higher-kinded type arguments presented a challenge; in Scala, they pose less of a problem. For example, the type of generalized trees can be defined as follows:

```
sealed case class GRose [F [_], A] (x : A, children : F [GRose [F, A]])
```

It is possible to define an implicit representation for the generic sum function using an approach similar to the *polymorphic predicates* technique, as discussed in Section 2.3.6.

```
implicit def totalGRose [F [_], A] (implicit totalA : Total [A],
  totalF : {def apply [B] (implicit totalB : Total [B]) : Total [F [B]]}) =
    new Total [GRose [F, A]] {
      def total = r ⇒ totalA.total (r.x) +
        totalF (totalGRose [F, A] (totalA, totalF)).total (r.children)
    }
```

However, Scala currently does not support type inference for datatypes with higher-kinded type arguments, so in the definition above it is necessary to explicitly construct the dictionary

```
totalF (totalGRose [F, A] (totalA, totalF))
```

so that the dictionary constructor function *totalGRose* can be passed the type constructor argument F. To emulate polymorphic predicates, such as the *totalF* argument of *totalGRose*, we have to encode higher-ranked types; this is not too hard using *structural types*, as discussed in more detail in Section 10.3. It is also necessary to provide a dictionary object that fits the interface required for *totalF* values:

```
implicit object totalList2 {
  def apply [a] (implicit totalA : Total [a]) : Total [List [a]] =
    totalList [a] (totalA)
}
```

Having set up all this machinery, we can now apply generic sum to generalized rose trees:

```
val myRose : GRose [List, Int] = GRose [List, Int] (3, Nil)

def test (rose : GRose [List, Int]) (implicit totalR : (Total [GRose [List, Int]])) =
  totalR.total (rose)

def testCount : Int = test (myRose) (totalGRose [List, Int] (countInt, totalList2))
```

The function *test* takes a value *rose* of type *GRose* [*List*, *Int*] and an implicit dictionary for the generic sum function, and returns the result of applying the appropriate member of this dictionary to the rose tree. The function *testCount* applies this function to a particular rose tree and an explicit dictionary for generalized rose trees, returning the result 1. Unfortunately, the dictionary cannot be constructed automatically, because the higher-kinded type *List* cannot be inferred.

6.4 Evaluation of the approach

Not surprisingly, the Scala approach has some additional verbosity when compared to the Haskell one, in particular with the long-winded syntax for implicits. In the Scala approach, a *gtotal* function is required in order to be able to pass a dictionary explicitly, while in Haskell no such definition is necessary. However, *gtotal* can be used for passing arguments both implicitly and explicitly, while in Haskell only implicit dictionaries are possible. Exotic types pose a challenge to Scala, as they do for Haskell, but for different reasons. In Scala, there is no native support for higher-ranked types; they have to be encoded, which adds extra overhead. Furthermore, the current version of Scala does not yet support type inference for higher-kinded types; in practice, this means that it is essentially not possible to automatically infer dictionaries that involve higher-kinded types. Having to write those dictionaries manually is tedious. However, in the Scala approach, local redefinition is possible, and can be used in a convenient way.

In summary, the Scala approach is slightly more verbose than the corresponding Haskell version, but it is also more expressive, since both local redefinitions and polymorphic predicates are possible. However, the latter feature has some significant overhead that makes it impractical to use.

7 Generic programming with encodings of datatypes

This section presents the first of three case studies on existing DGP approaches. Building on Section 4.5, which showed how implicit parameters can be used for type-class-style programming, a Scala implementation of the GM technique (Hinze 2006) is shown. Furthermore, we discuss two distinct techniques for reusing generic functions in Scala: *reuse by inheritance* and *local redefinition*. (Moors (2007) provides an alternative Scala tutorial on GM.)

7.1 Generics for the Masses, in Haskell

Figure 7 shows the essence of the GM approach in Haskell. The constructor class *Generic* is used to represent the type of generic functions. The parameter *g* represents the generic function, and each of the member functions of the class encodes the behaviour of that generic function for a specific structural case. Generic functions over user-defined types can be defined using the *view* type case: an isomorphism between the datatype and its structural representation must be provided. Instances of the type class *Rep* denote representable types; each such instance consists of a method *accept* that selects the appropriate behaviour from a generic function.

A new generic function is represented as an instance of *Generic*, providing an implementation for each structural case. For instance, consider a generic template for functions that compute some integer measure of a data structure. Each case is a record of type *Count a* for some type *a*, which contains a single function *count* of type $a \rightarrow Int$ that can be used for a structure of type *a*. The function *gCount*, which is the actual generic function, simply extracts the sole field *count* from a record of the appropriate type, built automatically by *accept*. For sums, products, and user-defined datatypes, it does 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; it counts zero for each of the base cases, but these can be overridden to implement more interesting behaviour.

```
class Generic g where
  unit     :: g Unit
  plus     :: g a → g b → g (Sum a b)
  prod     :: g a → g b → g (Prod a b)
  view     :: Iso b a → g a → g b
  char     :: g Char
  int      :: g Int

class Rep a where
  accept        :: Generic g ⇒ g a
instance Rep Unit where
  accept        = unit
instance Rep Char where
  accept        = char
instance Rep Int where
  accept        = int
instance (Rep a, Rep b) ⇒ Rep (Sum a b) where
  accept        = plus accept accept
instance (Rep a, Rep b) ⇒ Rep (Prod a b) where
  accept        = prod accept accept

newtype Count a = Count {count :: a → Int}

instance Generic Count where
  unit       = Count (λ_ → 0)
  char       = Count (λ_ → 0)
  int        = Count (λ_ → 0)
  plus a b   = Count (λx → case x of {Inl l → count a l; Inr r → count b r})
  prod a b   = Count (λ(Prod x y) → count a x + count b y)
  view iso a = Count (λx → count a (from iso x))

gCount :: Rep a ⇒ a → Int
gCount = count accept
```

Fig. 7. ‘Generics for the Masses’ in Haskell.

7.2 *Generics for the Masses, in Scala*

Figure 8 presents a translation of the code in Figure 7 into Scala. The trait *Generic* is parametrized with a higher-kinded type constructor G. As in Haskell, there are methods for sums, products, the unit type, and also a few built-in types such as integers and characters; 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 an isomorphism to adapt generic functions to existing datatypes; the ‘⇒’ before the type $G[a]$ signals that that parameter is passed by name.

The trait *Rep* $[T]$ has a single method *accept*, which takes an encoded generic function of type *Generic* $[G]$. The Scala implementation of the subclasses of *Rep* $[T]$ is almost a transliteration of the Haskell type class version, except that it uses implicit parameters instead of type classes.

The generic counter function uses a parametrized class *Count* with a single field: a function of type $A \Rightarrow Int$. The concrete subtype *CountG* of the trait *Generic* $[Count]$ provides implementations for the actual generic function: each method yields a value of type *Count* $[A]$ for the appropriate type A.

```
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]
}
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))
}
case class Count [A] (count : A ⇒ Int)
trait CountG extends Generic [Count] {
  def unit       = Count (x ⇒ 0)
  def int        = Count (x ⇒ 0)
  def char       = Count (x ⇒ 0)
  def plus [a,b] = a ⇒ b ⇒ Count (_.fold (a.count, b.count))
  def prod [a,b] = a ⇒ b ⇒ Count (x ⇒ a.count (x._1) + b.count (x._2))
  def view [a,b] = iso ⇒ a ⇒ Count (x ⇒ a.count (iso.from (x)))
}
```

Fig. 8. ‘Generics for the Masses’ in Scala.

7.3 Constructing type representations

For each datatype T we want to represent, we need to create a value of type *Rep* [T]. For example, for lists we could write:

```
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*; *listIso* is the isomorphism presented in Section 5.2.) Here, the auxiliary function *listRep* constructs the right *Generic* value following the sum-of-product structure. Using *listRep*, the representation *RList* for lists is easily defined.

7.4 Applying generic functions

We can now define a method *gCount* that provides an easy-to-use interface for the generic function encoded by *CountG*: this takes a value of a representable type *a* and returns the corresponding count.

```
def gCount [a] (x : a) (implicit rep : Rep [a]) = rep.accept [Count].count (x)
```

We defined *CountG* as a trait instead of an object so that it can be extended, as we discuss in more detail in Sections 7.5 and 7.6. We may, however, be interested in having an object that simply inherits the basic functionality defined in *CountG*. Furthermore, this object can be made implicit, so that methods like *rep* can automatically infer this instance of *Generic*.

```
implicit object countG extends CountG
```

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

7.5 Reuse via inheritance

To recover a generic function that counts the integers in a structure, we can use inheritance to extend *CountG* and override the case for integers so that it counts 1 for each integer value.

```
trait CountInt extends CountG {override def int = Count (x ⇒ 1)}
```

We can then define a method *countInt* to count the integers in any structure of representable type.

```
def countInt [a] (x : a) (implicit rep : Rep [a]) =
  rep.accept [Count] (new CountInt {}).count (x)
```

The ability to explicitly pass an alternative ‘dictionary’ is essential to the definition of the method *countInt*, since we need to parametrize the *accept* method with an instance of *Count* 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)
```

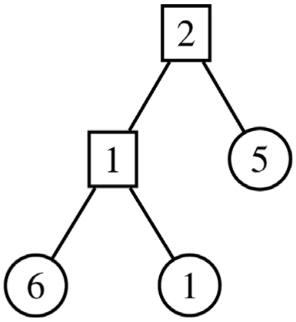

Fig. 9. Tree with depth information at the nodes.

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 def** *RList*).

7.6 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 to distinguish 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 in keeping the tree balanced. Figure 9 shows such a tree; the squares represent the auxiliary integers, and the circles represent the values contained in the tree. 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 binary trees; assuming that the equivalent method is called *btreeRep*, we can provide a special-purpose counter for trees that counts only the values of the type parameter.

```
def countOne [a] = Count ((x : a) ⇒ 1)
def countTree [a] (x : Tree [a]) = btreeRep [a, Count] (countOne [a]).count (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*.

7.7 Evaluation of the approach

Like other generic programming approaches, the GM technique is more verbose in Scala than in Haskell: in the definitions of instances (such as *RUnit*, *RChar*, and *RProd*) of the trait *Rep*, we need to explicitly declare 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 provides everything present in the Haskell solution, including the ability to handle local redefinitions. In addition, we can easily reuse one generic function to define another through inheritance, as demonstrated in Section 7.5; with the Haskell approaches, this kind of reuse is harder to achieve. The only mechanism that we know of that comes close to this form of reuse in terms of simplicity is Generic Haskell's *default case* construct (Löh 2004), as discussed in Section 2.3.5.

Another nice aspect of the Scala approach 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 7.5), we needed to override that argument. This was easily achieved in Scala simply 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 & Thielecke 2006; Oliveira 2007) – it does not seem to have 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 (Oliveira 2009a).

8 Generic programming with extensible superclasses

This section presents the second case study of a DGP library in Scala. We show how to emulate *extensible superclasses* (Sulzmann & Wang 2006), and how this technique can be used to provide an implementation of the *Scrap your Boilerplate with Class* (Lämmel & Peyton Jones 2005) approach to generic programming.

8.1 Scrap your boilerplate with class

After realizing that earlier implementations of the SyB approach (Lämmel & Peyton Jones 2003) were limiting because they did not support *extensible generic functions*,

```
data Proxy (a :: ∗ → ∗)

class Sat a where dict :: a

class (Typeable a, Sat (ctx a)) ⇒ Data ctx a where
  gmapQ :: Proxy ctx → (∀b. Data ctx b ⇒ b → r) → a → [r]

instance Sat (ctx Char) ⇒ Data ctx Char where
  gmapQ _ f n       = []

instance (Sat (ctx [a]), Data ctx a) ⇒ Data ctx [a] where
  gmapQ _ f []       = []
  gmapQ _ f (x : xs) = [f x, f xs]

class Size a where gsize :: a → Int

data SizeD a = SizeD {gsizeD :: a → Int}

sizeProxy :: Proxy SizeD
sizeProxy = ⊥

instance Size t ⇒ Sat (SizeD t) where
  dict = SizeD {gsizeD = gsize}

instance Data SizeD t ⇒ Size t where
  gsize t = 1 + sum (gmapQ sizeProxy (gsizeD dict) t)

instance Size a ⇒ Size [a] where
  gsize []       = 0
  gsize (x : xs) = gsize x + gsize xs

test = (gsize ['a','b'], gsize 'x')
```

Fig. 10. The original 'SyB with Class' implementation in Haskell.

Lämmel and Peyton Jones (2005) proposed a variation using type classes. This solution is shown in Figure 10. The *Data* class defines the higher-order generic function *gmapQ*, which is used to define new generic functions. The *Size* class declares a new generic function *size*. *Overlapping instances* are used to provide a default implementation of the function in terms of *gmapQ*; in the case of *Size*, the instance *Size t* plays this role. Generic functions can be made extensible by providing additional instances of class *Size* that override the default case. The solution is somewhat involved, and it requires a number of non-standard Haskell extensions to get everything to work. In particular, *undecidable instances* are needed, and an extension allowing *recursive dictionaries* had to be built into the GHC compiler. Also, proxies for types, which involve passing an extra (bottom) value to functions, are required to resolve type ambiguities.

The major difficulty Lämmel and Peyton Jones found was that, in order to provide a modular definition of a new generic function, the *Data* class had itself to be parametrized by the generic function being defined. In essence, what seems to be needed here are extensible superclasses. Inspired by Hughes' (1999) work on *restricted datatypes*, Lämmel and Peyton Jones found a solution by emulating type class parametrization: in the class *Data ctx a*, the *ctx* argument is supposed to

```
trait Data [ctx [_], a] {self : ctx [a] ⇒
  def me = self
  def gmapQ [r] : {def apply [b] (x : b) (implicit dt : Data [ctx, b]) : r} ⇒ a ⇒ List [r]
}
abstract case class DataChar [ctx [_]] () extends Data [ctx, Char] {self : ctx [Char] ⇒
  def gmapQ [r] = f ⇒ n ⇒ Nil
}
abstract case class DataList [ctx [_], a] (implicit d : Data [ctx, a]) extends Data [ctx, List [a]] {
  self : ctx [List [a]] ⇒
  def gmapQ [r] = f ⇒ {
      case Nil    ⇒ Nil
      case x :: xs ⇒ List (f (x) (d), f (xs) (this))
  }
}
trait Size [a] extends Data [Size, a] {
  def gsize : a ⇒ Int = t ⇒
    1 + sum (gmapQ [Int] (new {def apply [b] (x : b) (implicit dt : Data [Size, b]) =
      dt.me.gsize (x)}) (t))
}
abstract case class SizeList [a] () (implicit d : Size [a])
  extends DataList [Size, a] () (d) with Size [List [a]] {
    override def gsize = {
      case Nil ⇒ 0
      case x :: xs ⇒ d.gsize (x) + gsize (xs)
  }
}
implicit def sizeChar : Size [Char] =
  new DataChar [Size] () with Size [Char]
implicit def sizeList2 [a] (implicit d : Size [a]) : Size [List [a]] =
  new SizeList [a] () (d) with Size [List [a]]
def test (implicit s1 : Size [Char], s2 : Size [List [Char]]) =
  (s1.gsize ('a'), s2.gsize (List ('a', 'b')))
```

Fig. 11. An implementation of ‘SyB with Class’ in Scala.

represent an unknown type class, but because Haskell does not allow abstraction over type classes, this has to be emulated using records.

8.2 *Scrap your boilerplate with class, in Scala*

Because of the wide range of non-standard features of Haskell used by the SyB with Class approach, it is interesting to see what is involved in expressing the approach in Scala. Like Haskell, Scala does not support extensible superclasses directly; that is, it is not possible to have a trait (or class)

trait *T* [*Super*] **extends** *Super*

in which the trait is parametrized by its own superclass. However, Scala does provide *explicit self-types* (Odersky 2006a), which can be used to emulate this feature. In Figure 11, a Scala implementation of the *Data* class is shown. As with the Haskell solution, the *Data* trait is parametrized by a type constructor *ctx* (the generic function) and a type *a*. The major difference from the Haskell solution is the use of a *self-type* to ensure that the type of the **self** object is a subtype of *ctx* [*a*]. This is to make the generic functions defined in *ctx* available to all instances of *Data*. (The definition *me* is just a public reference for the self object, and the *gmapQ* generic function uses the technique discussed in Section 10.3 to emulate higher-ranked types.) Two base 'instances' are provided for characters and lists, as in the Haskell implementation. *Abstract case classes* are used because *DataChar* and *DataList* are incomplete, that is, they still need to be mixed with implementations of the types *ctx* [*Char*] and *ctx* [*List* [*a*]]. The trait *Size* extends *Data* and defines the generic function *gsize* in terms of *gmapQ*. This trait plays the role of both the *Size* class and the *Size t* instance in the Haskell solution. The abstract case class *SizeList* provides the overriding case for lists. Note that *Size* and *SizeList* satisfy, respectively, the *Data* [*Size*, *a*] and *DataList* [*Size*, *a*] requirements for the self-type. The implicit definitions *sizeChar* and *sizeList* allow the dictionaries for characters and lists to be built automatically. Finally, *test* shows how the generic function can be used – here, to compute the size of a character and of a list of characters. Because *test* takes two implicit arguments, it is possible to call it without those arguments; alternatively, different dictionaries can be provided, overriding the ones selected by the compiler.

8.3 Local redefinition

In the Scala implementation of SyB with Class, local redefinition is possible. For example, instead of using the *sizeList* dictionary for lists, it is possible to provide an alternative dictionary that inherits the generic behaviour for lists rather than overriding it:

```
def alternativeList [a] (implicit d : Size [a]) : Size [List [a]] =
  new DataList [Size, a] () (d) with Size [List [a]]
```

Given this definition, both *test* and *test* (*sizeChar*, *alternativeList*) are possible applications, returning (1, 2) and (1, 5) respectively.

8.4 Evaluation of the approach

Verbosity is once again a problem. The lack of direct support for higher-ranked types and a long-winded syntax for implicits adds significant additional code in comparison to the Haskell approach. Another problem is that separate implicit definitions for dictionaries like *sizeChar* and *sizeList* are needed.

The Scala approach imposes an additional burden on the programmer due to the absence of a mechanism similar to *overlapping instances*. This requires the programmer to implement the definitions for the implicit dictionaries one by one. In the Haskell solution, if there is no overridden case, then no additional effort is needed. On the other hand, the Scala solution does not distinguish between types and type classes, and abstracting over the 'type class' is just the same as abstracting over a type: no encoding of this feature is required. Furthermore, a solution with explicit self-types does not require other advanced features such as recursive dictionaries or undecidable instances; everything is accomplished naturally, using the standard extension mechanism.

In terms of expressiveness, the Scala solution is better, because it supports local redefinitions and allows greater control of dictionaries by providing the possibility to pass them explicitly. In the original SyB with 'Class' solution, local redefinitions are not possible.

In summary, for the SyB with 'Class' approach the results are mixed: Haskell is more convenient to use because it imposes a lighter burden on the programmer, but the Scala solution is more expressive and flexible because local redefinitions are possible.

9 Generic programming with recursion patterns

Most generic programming libraries involve writing generic functions by case analysis on the structure of the shape of the datatype, whether that case analysis is by value-based or type-based dispatch. An alternative is to make the shape parameter an active participant in the computation – a higher-order function that can be applied, rather than passive data that must be analyzed. In particular, the *Origami Programming* (Gibbons 2003) approach to DGP is based around datatypes represented as fixpoints of type functors, and programs expressed in terms of higher-order recursion patterns shape-parametrized by those functors (Meijer *et al.* 1991). A consequence of black-box application rather than white-box inspection of the shape parameter is a kind of higher-order naturality property, guaranteeing coherence between different instances of the generic function (Gibbons & Paterson 2009).

One can view the origami recursion patterns as functional programming equivalents to (at least the code aspects of) some of the so-called Gang of Four design patterns (Gamma *et al.* 1995). Gibbons (2006) argues that recursive datatypes correspond to the COMPOSITE design pattern, maps to the ITERATOR pattern for enumerating the elements of a collection, folds to the VISITOR pattern for traversing a hierarchical structure, and unfolds and builds to structured and unstructured instances of the BUILDER pattern for generating structured data.

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 these origami patterns in Scala. However, their encoding was done in an object-oriented style that introduced some limitations that the original Haskell version did not have. We feel that this object-oriented style, while perhaps more familiar to the object-oriented programmer that Moors *et al.* were targeting, does not show the full potential of Scala from a generic programmer's perspective. In this section, 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.

9.1 A little Origami library

Figure 12 shows the Haskell implementation of the origami patterns, and Figure 13 shows a translation of this Haskell code into Scala. The key idea is to encode type

```
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 :: (∀b. (f a b → b) → b) → Fix f a
build f = f In
```

Fig. 12. Origami in Haskell.

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

Fig. 13. Origami in Scala.

classes through implicit parameters (see Section 4.5) rather than using the object-oriented style proposed by Moors *et al.*. The newtype *Fix* and its constructor *In* are mapped into a case class *Fix*; 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 two 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)
```

```
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))
```

Fig. 14. Lists as a fixpoint.

(the syntax $(\_.m)$ is syntactic sugar for $(x \Rightarrow x.m)$; in other words, '_' denotes an 'anonymous' lambda variable). Under strict evaluation, the above definition would expand indefinitely; we have to write it less elegantly using application rather than composition. Secondly, higher-ranked types are once again required; we have to encode them in Scala – see Section 10.3 for more details.

9.2 Using the library

Figure 14 captures the shape of lists as a type constructor *ListF*; the two possible shapes for lists are defined with the case classes *Nil* and *Cons*. The *BiFunctor* object *biList* 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
}
```

9.3 Evaluation of the approach

Figure 15 presents Moors *et al.*'s object-oriented encoding of the origami operators (slightly adapted due to intervening changes in Scala syntax), and Figure 16 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

```
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
}
```

Fig. 15. Origami in Scala, after Moors *et al.*

```
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]
```

Fig. 16. Lists as a fixpoint, after Moors *et al.*

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.

The dependence on the 'self' parameter in the OO encoding requires *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 elaborate construction for guaranteeing type safety; it is not required at all in the FP encoding.

A second characteristic of the OO encoding is the way operations are attached to objects as methods; for example, *cata* is a method 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 does not 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 does not 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 & Jeuring 1995), paramorphism (Meertens 1992), or idiomatic traversal (Gibbons & Oliveira 2009), 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 syntactically rather more noisy than Haskell, for a variety of reasons: the use of parentheses rather than simple juxtaposition for function application; explicit binding of type variables, for example in indicating that *bimap* is parametrized by the four types a, b, c, d; the lack of eta reduction because of call-by-value, 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 the language.

On the positive side, the translation is quite direct, and the encoding rather transparent; the code in Figure 13 is not that much more intimidating than that in Figure 12. 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 '$\lambda x \rightarrow$ **case** x **of** ...'.

10 Discussion

10.1 Haskell versus Scala

Scala differs significantly from Haskell, and we were curious to know what were its advantages and disadvantages when implementing generic programming libraries.

This work was done using the Glasgow Haskell Compiler version 6.10 and Scala version 2.7, which were the latest official releases at the time of writing. However, the languages will keep evolving, and in the future it is likely that both languages will provide better support for generic programming. Indeed, the next version (2.8) of the Scala compiler will support a few features that could have been useful for our work: *context bounds*, which provide a compact syntax for implicits; *prioratized overlapping implicits*, which provide an alternative to overlapping instances; and *type-inference for type constructors*. However, for consistency with the results presented in this paper, we shall not consider these features in the discussion that follows.

Generally speaking, Haskell has a few advantages over Scala:

Laziness: Some approaches to generic programming rely, one way or another, on laziness. While laziness comes without effort in Haskell, it does not in Scala, and we need to pay more attention to evaluation order: we had to adapt the origami definitions in Section 9, and introduce call-by-name arguments in the *RView* constructor in Figure 5.

Type inference: Haskell has good support for type inference, which helps to reduce the effort and clutter demanded by generic programming libraries. Scala's support for type inference is not as good, and this leads to additional verbosity and complexity of use.

Syntactic clarity: While Scala's syntax is more elegant than that of Java or C#, it is still more verbose than Haskell's. In particular, we have to declare more types in Scala, and need to write extra type annotations. Also, the syntax for implicits can be a bit unwieldy, and case classes can be slightly more cumbersome than Haskell's **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 carefully control 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 (except for non-termination); Scala provides no such guarantees.

Higher-ranked types: Some implementations of Haskell provide support for higher-ranked types, while in Scala they need to be encoded. Because higher-ranked types play a role in some aspects of DGP, the additional overhead required by the encoding can be a significant drawback.

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 & Scheffczyk 2001; Dijkstra & Swierstra 2005). The ability to override implicit dictionaries is a desirable feature for generic programming (Löh 2004, Chapter 8).

	…Haskell…		*…Scala…*	
	Datatypes	*Type classes*	*Case classes*	*Standard classes*
Convenience:				
Defining generic functions	●	◐	●	◐
Using generic functions	◐	●	●	●
Implicit explicit parametrization	○	○	●	●
Extensibility	○	●	●	●
First-class generic functions	●	○	◐	◐
Reuse of generic functions	○	○	●[1]	◐[2]
Exotic types	●	◐	◐	◐

Fig. 17. Evaluation of the Haskell mechanisms for DGP. Key: ●='good', ◐='sufficient', ○='poor' support. Notes: (1) generic functions need to be written using classes rather than function definitions; (2) reuse can be achieved in Scala via inheritance with virtual types.

Inheritance: Another advantage of Scala is that we can easily reuse generic functions via inheritance. In Haskell, although we can simulate this form of reuse in several ways, there is no natural way to do so.

Expressive type system: The combination of subtyping, higher kinds, abstract types, implicit parameters, traits and mixins (among other features) provides Scala with an impressively powerful type system. Although we do not fully exploit the expressivity in this paper, 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 9.3) quite neat and useful. Although we seldom need to provide type annotations in Haskell expressions, they can be quite tricky to get right when they are needed; in Scala this is easier. 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 7.7).

10.2 Support for DGP in Scala and Haskell

The most noticeable difference between the Haskell and Scala approaches to DGP is that type classes and datatypes are essentially two separate mechanisms in Haskell; in contrast, in Scala, the same mechanism – Scala's object system – is used, albeit in different ways, to define standard OO hierarchies and algebraic datatype-like structures.

Figure 17 extends the table presented in Figure 2 to include the approaches presented in Sections 5 and 6, which can be considered to be the equivalents of the Haskell approaches in Scala. Specifically, case classes are used to implement the datatype approach in Scala, while standard OO classes (with implicits) are used to implement the type class approach. We discuss and summarize the results in the table for the Scala approaches next.

Convenience. Defining and using generic functions with case classes is quite natural, so this mechanism scores 'good' for both aspects of convenience. Compared to

Haskell, in Scala there is an advantage of using datatypes of type representations because the value of the type representation can be implicit, whereas in Haskell (without resorting to type classes) it has to be explicit. Using standard classes and implicits to implement the type class based approach confers no advantage over Haskell in terms of convenience. Approaches based on both Haskell's type classes and Scala's standard classes score only 'sufficient' for defining generic functions, since there is additional overhead compared to using a datatype of type representations.

Implicit explicit parametrization. The Scala approaches do well in this respect because of the implicits mechanism, which allows values to be passed implicitly or explicitly. In Haskell, the choice of mechanism determines the choice of parametrization: datatypes require explicitly passed values, whereas type classes require implicitly passed dictionaries. In other words, unlike in Haskell, implicit or explicit parametrization in Scala is independent of the particular mechanism chosen for implementing the DGP library.

Extensibility. This is another area in which Scala does well. As with the Haskell type class approach, using Scala classes to define generic functions provides extensibility by default. However, unlike Haskell, the datatype of type representations in Scala can also be extensible, since case classes are open. Furthermore, the case class mechanism provides a safer alternative to open datatypes and functions, preserving the advantages of static typing and avoiding pattern match failures by using sealed classes.

First-class generic functions. In this area, the results are mixed. On the one hand, Scala does support first-class generic functions, and it is possible to abstract over the type of the generic function directly in a type-class-based approach. On the other hand, Scala does not provide native support for higher-ranked types, which adds complexity and verbosity to generic functions. For this reason, Scala only scores 'sufficient'.

Reuse of generic functions. Scala does well here in comparison to Haskell: inheritance supports reuse of generic functions quite naturally. In the case-class approach, this support is quite direct, and can be used effectively to define new generic functions by inheriting from existing ones. A small inconvenience, though, is that we need to write function definitions using classes in order to be able to exploit inheritance. It is also possible to use inheritance to achieve reuse using the standard classes approach, but nested types and virtual types are required. Such a solution is not presented here, but is shown by Oliveira (2007, Chapter 5). However, this latter solution is rather involved and heavyweight, which hinders usability. Ultimately, we think that support for inheritance is helpful for generic programming, and that Scala is worthy of full marks for the case class approach.

Exotic types. This is an area in which Haskell is, for the most part, better than Scala. The two main reasons are Haskell's better support for type inference and for higher-ranked types. Because of that support, exotic types can be used cleanly with the datatype of type representations approach. In contrast, Scala's solution is hindered by the additional verbosity required due to the lack of native support for higher-ranked types and the less complete type inference. The standard classes approach in Scala has the merit of directly supporting the solution proposed by Hinze and Peyton Jones (2000), but like the other Scala solution the cost in terms of usability is quite high. Therefore, in this area, Scala only scores 'sufficient'.

10.3 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 3.5, in a definition like

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

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. A simple trick can help the compiler (at least to try) to infer argument types: use lambda expressions rather than passing parameters. That is, transform a parametrized method:

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

into a parameterless method with a higher-order value:

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

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

$$\textbf{def}\ f_T = x_1 \Rightarrow \ldots \Rightarrow x_n \Rightarrow e$$

The main difference between f and f_T is that the former can be (name) overloaded, while the latter cannot. As discussed in Section 3.5, name-overloaded definitions pose a challenge to type-inference. This transformation is used a few times to make the most of type inference, avoiding cluttering definitions with redundant type annotations; see for example the methods of *Generic* in Figure 8, and *bimap* in Figure 13.

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 Haskell type class *Ord*

```
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 OO approach. As we saw in Section 9, 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 & 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 :: ∀a.(∀b.b → b) → a → a
```

is encoded in Scala as:

```
def func [a] : {def apply [b] : b ⇒ b} ⇒ a ⇒ a
```

The type {**def** *apply* [*b*] :*b* ⇒ *b*} stands for *some* class with a method *apply* [*b*] :*b* ⇒ *b*. Structural types allow a definition that is nearly as short as the Haskell one. 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

```
def func [a] : {def apply [b <: a] : b ⇒ b} ⇒ a ⇒ 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 has been observed in the literature. We believe that providing primitive support for higher-ranked types in Scala using this encoding as a basis should be fairly simple.

Functional inheritance. In Scala, all functions are objects and, as such, are amenable to inheritance when it comes to reuse. Unfortunately, while Scala does support the conventional notation of function definition, this notation does not support inheritance. A definition like:

```
def func (x : Int) : Int = e
```

(where e is an expression that may depend on x) needs to be rewritten as:

```
case class Func extends (Int ⇒ Int) {
  def apply (x : Int) = e
}
```

Provided that functions are written in this style, then inheritance allows the reuse of function definitions, as demonstrated in Section 5.3.

10.4 Porting generic programming libraries to Scala

There has been a flurry of recent proposals for generic programming libraries in Haskell (Cheney & Hinze 2002; Lämmel & Peyton Jones 2005; Hinze 2006; Hinze *et al.* 2006; Oliveira *et al.* 2006; Weirich 2006; Hinze & Löh 2007; Mitchell & Runciman 2007; Brown & Sampson 2009), 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 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; one example is the approach discussed in Section 8. As we have seen, it is possible to implement such an approach in Scala, but the lack of support for type inference for higher kinds and something like *overlapping instances* means that additional explicitness and effort is required in Scala. Therefore, approaches that make intensive use of advanced type class features can be ported, but they may lose some usability in Scala.

Something that Scala does not have is a meta-programming facility. Some of the generic programming libraries use *Template Haskell* (Sheard & 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 & 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 instances manually.

11 Conclusions

The goal of this paper was not to promote a particular approach to generic programming. Instead, we were more interested in investigating how the language mechanisms of Haskell used in various generic programming techniques could be adapted to Scala. We hope that this work can serve as a foundation for future development of generic programming libraries in Scala: all of the approaches discussed in this paper could serve as good starting points for more complete libraries. Moreover, other approaches can still benefit from the discussions we present.

As we have argued, Scala has some features that are very useful in a datatype-generic programming language. We expect that other programming languages (in particular, Haskell) can learn some lessons from Scala by borrowing these features. Conversely, Haskell has some features useful for DGP that are not available in Scala, but which would be nice to have. Ultimately, we believe that we have pinpointed limitations of some general-purpose language mechanisms for implementing DGP libraries; hopefully, this will motivate the development of improved mechanisms or programming languages. Oliveira and Sulzmann (2008) have already done some preliminary work in that direction by proposing a generalized class system for a Haskell-like language that is partly inspired by Scala, and which allows both implicit and explicit passing of dictionaries.

Acknowledgments

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 insightful advice and constructive comments. Adriaan Moors made many useful suggestions; and the members of the Scala mailing list were very helpful in answering some questions related to this paper – their input, and guidance from the anonymous reviewers of the *Workshop on Generic Programming* in 2008 and of this journal, helped to improve the presentation. The work reported in this paper has been supported by the UK Engineering and Physical Sciences Research Council grant *Generic and Indexed Programming* (EP/E02128X) and 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. Some of the work was conducted while the first author was at Oxford University Computing Laboratory.

References

Agrawal, R., Demichiel, L. G. & Lindsay, B. G. (1991) Static type checking of multi-methods. In *Proceedings of Object Oriented Programming Systems Languages and Applications*. Phoenix, AZ, USA, pp. 113–128.

Brown, N. C. C. & Sampson, A. T. (2009) Alloy: Fast generic transformations for Haskell. In *Haskell Symposium*. Edinburgh, pp. 105–116.

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

Buchlovsky, P. & Thielecke, H. (2006) A type-theoretic reconstruction of the Visitor pattern, *Electron. Notes Theor. Comput. Sci.*, 155, 309–329 (Mathematical Foundations of Programming Semantics).

Cheney, J. & Hinze, R. (2002) A lightweight implementation of generics and dynamics. In *Haskell Workshop*. Pittsburgh, pp. 90–104.

Cockett, R. & Fukushima, T. (May 1992) *About Charity*. Department of Computer Science, University of Calgary.

Cook, W. R. (1989) *A denotational Semantics of Inheritance*. Ph.D. thesis, Brown University.

Dijkstra, A. & Swierstra, S. D. (2005) *Making Implicit Parameters Explicit*. Tech. rept. UU-CS-2005-032. Department of Information and Computing Sciences, Utrecht University.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gibbons, J. (2003) Origami programming. In *The Fun of Programming*, Gibbons, J. & de Moor, O. (eds), Palgrave Macmillan, pp. 41–60.

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

Gibbons, J. & de Moor, O. (eds) (2003) *The Fun of Programming*. Cornerstones in Computing Serie. Palgrave Macmillan.

Gibbons, J. & Oliveira, B. C. D. S. (2009) The essence of the Iterator pattern, *J. Funct. Program.*, 19, 377–402.

Gibbons, J. & Paterson, R. (2009) Parametric datatype-genericity. In *Workshop on Generic Programming*, Jansson, P. & Schupp, S. (eds). Edinburgh, pp. 85–93.

Hall, C. V., Hammond, K., Peyton, J., Simon, L. & Wadler, P. L. (1996) Type classes in Haskell, *ACM Trans. Program. Lang. Syst.*, 18 (2), 109–138.

Harper, R. & Lillibridge, M. (January 1994) A type-theoretic approach to higher-order modules with sharing. In *Principles of Programming Languages*, pp. 123–137.

Hinze, R. & Peyton Jones, S. (2000) Derivable type classes, In *Haskell Workshop: Electronic Notes in Theoretical Computer Science*, 41 (1), 5–35.

Hinze, R. (2000) Polytypic values possess polykinded types. In *LNCS 1837: Proceedings of the Fifth International Conference on Mathematics of Program Construction*, Backhouse, R. & Oliveira, J. N. (eds). Springer-Verlag, pp. 2–27.

Hinze, R. (2003) Fun with phantom types. In *The Fun of Programming*, Gibbons, J. & de Moor, O. (eds), Palgrave Macmillan.

Hinze, R. (2006) Generics for the masses, *J. Funct. Program.*, 16 (4–5), 451–483.

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

Hinze, R. & Löh, A. (2007) Generic programming, now! *LNCS 4719: Datatype-Generic Programming*.

Hinze, R. & Löh, A. (2009) Generic programming in 3D, *Sci. Comput. Program.*, 74 (8), 590–628.

Hinze, R., Löh, A. & Oliveira, B. C. D. S. (2006) 'Scrap your Boilerplate' reloaded. *LNCS 3945: Functional and Logic Programming*, pp. 13–29.

Hughes, J. (1999) Restricted data types in Haskell. Meijer, E. (ed), In *Haskell Workshop*. Paris.

Jansson, P. (May 2000) *Functional Polytypic Programming*. Ph.D. thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden.

Kahl, W. & Scheffczyk, J. (2001) Named instances for Haskell type classes. In *Haskell Workshop*. Firenze, pp. 77–99.

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

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

Lämmel, R., Visser, J. & Kort, J. (July 2000) Dealing with large bananas. In *Workshop on Generic Programming*, Jeuring, J. (ed), pp. 46–59.

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

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

Löh, A. (2004) *Exploring Generic Haskell*. Ph.D. thesis, Utrecht University.

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

McBride, C. & Paterson, R. (2008) Applicative programming with effects, *J. Funct. Program.*, 18 (1), pp. 1–13.

Meertens, L. (1992) Paramorphisms, *Form. Asp. Comput.*, 4 (5), 413–425.

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

Meijer, E., Fokkinga, M. & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. Hughes, J. (ed), In *LNCS 523: Functional Programming Languages and Computer Architecture*, pp. 124–144.

Mitchell, N. & Runciman, C. (2007) Uniform boilerplate and list processing. In *Haskell Workshop*, pp. 49–60.

Moors, A. (June 2007) *Code-follows-type Programming in Scala* [online]. Available at: `http://www.cs.kuleuven.be/~adriaan/?q=cft_intro`

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

Moors, A., Piessens, F. & Odersky, M. (2008) Generics of a higher kind, *Object-Oriented Program. Syst. Lang. Appl.*.

Odersky, M. (2006a) *An Overview of the Scala Programming Language (second edition)*. Tech. rept. IC/2006/001. EPFL Lausanne, Switzerland.

Odersky, M. (July 2006b) *Poor Man's Type Classes* [online]. Available at: `http://lamp.epfl.ch/~odersky/talks/wg2.8-boston06.pdf`

Odersky, M. (May 2007a) *Scala by Example* [online]. Available at: `http://scala.epfl.ch/docu/files/ScalaIntro.pdf`

Odersky, M. (May 2007b) *The Scala Language Specification, version 2.4* [online]. Available at: `http://scala.epfl.ch/docu/files/ScalaReference.pdf`

Odersky, M. & Läufer, K. (1996) Putting type annotations to work. In *Principles of Programming Languages*, pp. 54–67.

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

Odersky, M., Zenger, C. & Zenger, M. (2001) Colored local type inference. In *Principles of Programming Languages*, pp. 41–53.

Odersky, M., Spoon, L. & Venners, B. (2008) *Programming in Scala: A Comprehensive Step-by-Step Guide*. 1st ed. Artima.

Oliveira, B. & Gibbons, J. (2005) TypeCase: a design pattern for type-indexed functions. In *Haskell Workshop*. New York, NY, USA: ACM Press, pp. 98–109.

Oliveira, B. C. D. S. (2007) *Genericity, Extensibility and Type-Safety in the* VISITOR *Pattern*. DPhil thesis, University of Oxford.

Oliveira, B. C. D. S. (July 2009a) Modular visitor components: A practical solution to the expression families problem. Drossopoulou, S. (ed), In *23rd European Conference on Object Oriented Programming (ECOOP)*, pp. 269–293.

Oliveira, B. C. D. S. (September 2009b) *Scala for Generic Programmers: Source code* [online]. Available at: `http://www.comlab.ox.ac.uk/projects/gip/Scala.tgz`

Oliveira, B. C. D. S. & Sulzmann, M. (April 2008) *Objects to Unify Type Classes and GADTs* [online]. Available at: `http://www.comlab.ox.ac.uk/people/Bruno.Oliveira/objects.pdf`

Oliveira, B. C. D. S., Hinze, R. & Löh, A. (April 2006) Extensible and modular generics for the masses. In *Trends in Functional Programming*, pp. 109–138.

Oliveira, B. C. D. S., Moors, A. & Odersky, M. (October 2010) Type classes as objects and implicits. In *Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*, Rinard, M. (ed).

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

Rodriguez, A., Jeuring, J., Jansson, P., Gerdes, A., Kiselyov, O. & Oliveira, B. C. D. S. (2008) Comparing libraries for generic programming in Haskell. In *Haskell Symposium.*

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

Schinz, M. (May 2007) *A Scala Tutorial for Java Programmers* [online]. Available at: `http://scala.epfl.ch/docu/files/ScalaTutorial.pdf`

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

Sulzmann, M. & Wang, M. (2006) Modular generic programming with extensible superclasses. In *Workshop on Generic Programming*. New York, NY, USA: ACM, pp. 55–65.

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

Wadler, P. (November 1998) *The Expression Problem.* Java Genericity Mailing list [online]. Available at: `http://www.cse.ohio-state.edu/~gb/cis888.07g/java-genericity/20`

Washburn, G. (May 2008) *Revisiting Higher-Rank Impredicative Polymorphism in Scala* [online]. Available at: `http://existentialtype.net/2008/05/26/revisiting-higher-rank-impredicat%ive-polymorphism-in-scala/`

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