Scrap your boilerplate with class: extensible generic functions

Ralf Lämmel

Microsoft Corp.
ralfla@microsoft.com

Abstract

The 'Scrap your boilerplate' approach to generic programming allows the programmer to write generic functions that can traverse arbitrary data structures, and yet have type-specific cases. However, the original approach required all the type-specific cases to be supplied at once, when the recursive knot of generic function definition is tied. Hence, generic functions were *closed*. In contrast, Haskell's type classes support *open*, or extensible, functions that can be extended with new type-specific cases as new data types are defined. In this paper, we extend the 'Scrap your boilerplate' approach to support this open style. On the way, we demonstrate the desirability of abstraction over type classes, and the usefulness of recursive dictionaries.

Categories and Subject Descriptors D.1.m [*Programming Techniques*]: Generic Programming; D.3.3 [*Programming Languages*]:

Language Constructs and Features; D.2.13 [*Software Engineering*]: Reusable Software

Keywords Generic programming, type classes, extensibility, typecase, recursive dictionaries

1. Introduction

In the so-called "scrap your boilerplate" approach to generic programming, we exploit Haskell's rich type system to allow programmers to write "generic" functions [LP03, LP04]. The approach works very well for constructing *closed* generic functions; that is, ones whose special cases are all known in advance. However, until now, the approach did not work well for *open*, or extensible, generic functions.

We consider a generic programming example to illustrate the open/closed dichotomy. The QuickCheck library [CH00] involves the following function:

```
shrink :: Shrink a => a -> [a]
```

Shrinking a data structure returns a list of smaller data structures of the same type. QuickCheck runs the user's function on randomly

ICFP'05 September 26–28, 2005, Tallinn, Estonia.

Simon Peyton Jones

Microsoft Research

simonpj@microsoft.com

chosen inputs. When it finds a value that fails a test, it repeatedly uses `shrink` to try to find a smaller example that also fails.

Shrinking is clearly a generic programming problem. For many data structures, a boilerplate definition will do, e.g., return the largest (immediate or deeply nested) subterms of the same type as the failing term. But some data structures require special treatment. For example, we must not shrink a syntax tree representing a program in such a way that variables become unbound.

Each user of the QuickCheck library defines new data types. So QuickCheck cannot define, once and for all, all the types for which `shrink` behaves specially; `shrink` absolutely must be extensible. That is not possible using the existing "scrap your boilerplate" approach, as Koen Claessen carefully explained to us[1]. In general terms, lack of open, generic functions effectively bans generic programming from use in libraries.

Thus motivated, this paper describes a variant of "scrap your boilerplate" (henceforth SYB) that directly supports open, generic functions. We make the following contributions:

- We describe how to program extensible generic functions in Haskell (Section 3). It was entirely non-obvious (at least to us)

that SYB could be enhanced in a such a manner.

- Our initial presentation assumes that Haskell allows abstraction over *type classes* in addition to normal abstraction over *types*. In particular, we need to parameterise a class by its superclass — a feature somewhat reminiscent of mixins. In Section 4 we build on work by Hughes to show that this extension is not necessary [Hug99]. However, we argue that abstraction over type classes is a natural and desirable extension — after all, Haskell lets you abstract over practically anything else.

- While our new approach builds on Haskell's type-class system — hence the title — it requires one fundamental extension, which we deliver in this paper: the ability to construct recursive dictionaries (Section 5). This extension is both principled and independently useful. It has been requested many times by (hard-core) Haskell users, and was already part of GHC before we began work on this paper.

We give a case study of the approach applied to QuickCheck in Section 6, and discuss related work in Section 8. Everything we describe has been implemented, as Haskell code that runs in GHC, and is available at `http://www.cs.vu.nl/boilerplate/`. The extended SYB is finding its way into new applications of generic programming such as Foster's HAIFA ("Haskell Application Interoperation Framework Architecture") [Fos05].

[1] Personal communication, October 2004.

2. The problem we tackle

Let's consider a very simple generic function that computes the size of a data structure:

```
gsize :: Data a => a -> Int
gsize t = 1 + sum (gmapQ gsize t)
```

Here we use the SYB combinator `gmapQ`, a method of the `Data` class, defined thus:

```
class Typeable a => Data a where
  gmapQ :: (forall b. Data b => b -> r)
        -> a -> [r]
```

The idea is that `(gmapQ gsize t)` applies `gsize` to each of the immediate children of `t`, and returns a list of these sizes; then `sum` adds up this list, and we conclude by adding 1 (for the root node) to the total. Instances of the `Data` class can be derived automatically, but we give two sample instances as an illustration:

```
instance  Data Char where
  gmapQ f c = []
   -- no immediate subterms to be queried

instance Data a => Data [a] where
  gmapQ f []     = []
   -- no immediate subterms to be queried
  gmapQ f (x:xs) = [f x, f xs]
   -- head and tail are queried
```

The `Data` class has several other methods, but for much of this paper we will pretend that it has just one method, `gmapQ`. Everything we say extends to generic function types other than just queries (c.f. "Q" in `gmapQ`).

2.1 Classic customisation

Almost always, however, one wants to define special cases of a generic function at specific types. For example, suppose that the datum `t` contained nodes of type `Name`:

```
data Name = N String deriving( Typeable )
```

Then we might want to count just 1 for a `Name` node, rather than count up the size of the string inside it. As another example, what would you expect the call `(gsize [4,1])` to return? In fact it returns 5, one for each cons cell, one for the nil at the end of the list, and one for each `Int`; but we might prefer to give `gsize` list-specific behaviour, so that it returned (say) the length of the list.

The original SYB paper [LP03] described how to achieve type-specific behaviour, using type-safe cast and operations defined on top of it. The main function, `gsize`, is obtained by combining a generic function `gsize_default` with a type-specific case, `name_size`, written by the programmer:

```
gsize :: Data a => a -> Int
gsize t = gsize_default `extQ` name_size
                        `extQ` phone_size

gsize_default :: Data a => a -> Int
gsize_default t = 1 + sum (gmapQ gsize t)

name_size :: Name -> Int
name_size (N _) = 1

phone_size :: PhoneNumber -> Int
-- Another special case
```

The type of the combinator `extQ`[2] is the following:

[2] "ext" hints at generic function *ext*ension — another term for customisation.

```
extQ :: (Typeable a, Typeable b)
     => (a->r) -> (b->r) -> (a->r)
```

Here, `Typeable` is a superclass of `Data`. In the call `(extQ f g t)`, `extQ` attempts a cast to decide whether to apply `g` to `t`, or to use the generic method `f`. Since `extQ` is left-associative, one can compose together a whole string of calls to `extQ` to give the function many type-specific cases.

2.2 The shortcomings of `extQ`

However, this way of specialising, or customising, a generic function suffers from several shortcomings:

- The cast operation of `extQ` boils down to a run-time type test. When a customised generic function is applied to a datum, then type tests are performed in linear sequence for the type-specific cases, at every node of a traversed data structure. These type tests can outweigh other computations by a factor.
- There is no static check for overlap; in a long sequence of `extQ` calls one could mistakenly add two cases for `Name`, one of which would silently override the other.
- The use of cast operations becomes fiddly when we want to specialise the generic function for type *constructors* as well as *types* [LP04]. A good example is when we want to specialise `gsize` for polymorphic lists, as suggested above.

But these problems pale into insignificance beside the main one:

- *Once the "knot" is tied, via the mutual recursion between* `gsize` *and* `gsize_default`*, one can no longer add type-specific cases to* `gsize`. Notice the way that `gsize` contains a list of all its type-specific cases.

In short, the technique is fundamentally non-modular. Suppose a programmer adds a new type `Boo`, and wants to extend `gsize` to handle it. The only way to do so is to tie the knot afresh:

```
my_gsize :: Data a => a -> Int
my_gsize t = gsize_default `extQ` name_size
                           `extQ` phone_size
                           `extQ` boo_size

gsize_default :: Data a => a -> Int
gsize_default t = 1 + sum (gmapQ my_gsize t)

boo_size :: Boo -> Int
...
```

The amount of new code can be reduced in obvious ways — for example, pass the recursive function to `gsize_default` as an argument, rather than calling it by name — but it still forces the programmer to explicitly gather together all the type-specific cases, and then tie the knot.

2.3 What we want

What makes the situation particularly tantalising is the contrast with type classes. In Haskell, if we declare a new type `Name`, we can extend equality to work over `Name` simply by giving an instance declaration:

```
instance Eq Name where
  (N s1) == (N s2) = s1==s2
```

The type system checks that there is only one instance for `Eq Name`. There is no run-time type test; instead, the correct instance is automatically selected based on static type information. If a function is polymorphic in a type with equality, then the correct instance can-

not be selected statically, so it is passed as a run-time parameter instead. For example:

```
isRev :: Eq a => [a] -> [a] -> Bool
isRev xs ys = (xs == reverse ys)
```

We know statically that the equality test is performed on two lists, but the element type of the lists is not known statically — hence the `(Eq a)` constraint in the type. At run-time, `isRev` is passed a "dictionary" that gives the equality method for values of type `a`, and from which it can construct the equality method for lists of type `[a]` (again by plain dictionary passing).

Most importantly, though, the programmer never has to gather together all the instances and define a recursive `==` that takes all these instances into account. The result is modular: each time you define a new type, you also define its overloaded operations.

Unfortunately, overloaded operations (in the Haskell sense) are not generic; you have to define an instance for *every* type. We want the best of both worlds: generic functions (in the scrap-your-boilerplate sense) together with modular customisation as new data types are added.

3. The idea

Our goal is to combine SYB with the modular extension offered by type classes. The pattern we hope to use is this:

- Each time we need a new generic function, such as `gsize`, we define a new type class, `Size`, with `gsize` as a method.
- At the same time, we provide a generic implementation of `gsize`, in the form of an instance for `(Size t)`. (Section 3.5 discusses an alternative.)
- When we later introduce a new data type, such as `Name` in the example above, we can also add an `instance` declaration for

`Size` that gives the type-specific behaviour of `gsize` for that type. If we omit such as specific instance, we simply inherit the generic behaviour.

It is helpful to identify three separate protagonists. The *SYB authors* (i.e., ourselves) write SYB library code, including the definition of the `Data` class and its supporting libraries. The *generic function author* writes another library that gives the class and generic definitions; in the case of `gsize`, this means the class `Size` and the generic definition of `gsize`. Finally the *client* imports this library, defines new types, and perhaps adds instance declarations that make `gsize` behave differently on these new types.

3.1 A failed attempt

Here is a first attempt:

```
class Size a where
  gsize :: a -> Int

instance Size Name where
  gsize (N _) = 1

instance Size t where
  gsize t = 1 + sum (gmapQ gsize t)
```

The idea is that the `Size Name` instance gives the `Name`-specific behaviour while the `Size t` instance gives the default, generic behaviour on all types that do not match `Name`. The reader will notice right away that this assumes that the compiler accepts overlapping instances, a non-standard extension to Haskell. Overlapping instances are very convenient here, but they are not absolutely necessary, as we discuss in Section 3.5. For now, however, let us assume that overlapping instances are allowed.

Overlap is not the big problem here. The problem is that the

`Size t` instance does not type-check! Recall the type of `gmapQ`:

```
gmapQ :: Data a => (forall b. Data b => b -> r)
                -> a -> [r]
```

There are two issues. First, the call to `gmapQ` in the `Size t` instance leads to a `Data t` constraint. So we must add `Data t` to the context of the instance declaration:

```
instance Data t => Size t where
  gsize t = 1 + sum (gmapQ gsize t)
```

The second issue is not so easily solved. In any call `(gmapQ f t)`, *the function* `f` *has access to the operations of the* `Data` *class (and its superclasses), but no more* — just look at the type of `gmapQ`. Sadly, in the `Size t` instance declaration we pass `gsize` to `gmapQ`, and `gsize` now has this type:

```
gsize :: Size a => a -> Int
```

The only obvious way out of this difficulty is to arrange that `Size` is a superclass of `Data`:

```
class (Typeable a, Size a) => Data a where ...
```

We have thus defined a single, extensible generic function[3].

3.2 Abstraction over a class

Problem solved? By no means. The `Data` class is defined in the SYB library, and we cannot extend it with a new superclass every time we want a new generic function! That would be a new (and even more pernicious) form of non-modularity. However, it leads us in an interesting new direction. Since we do not know what class should be a superclass of `Data`, *let us parameterise over that class*:

```
-- Pseudo-code
```

```
class (Typeable a, cxt a) => Data cxt a where
  gmapQ :: (forall b. Data cxt b => b -> r)
        -> a -> [r]

instance Data Size t => Size t where
  gsize t = 1 + sum (gmapQ gsize t)
```

Here the variable `cxt` ranges over *type classes*, not over *types*. In the class declaration for `Data`, the superclass is not fixed, but rather is specified by `cxt`. In the generic instance declaration for `Size t` we specify which particular superclass we want, namely `Size`.

We note that Haskell does not offer variables that range over type classes, but we will assume for now that it does. In Section 4.1 we will show how class parameters can be encoded straightforwardly in standard Haskell.

We are nearly home, but not quite. Let us recall again the types for `gmapQ` and `gsize`, which we write with fully-explicit quantification:

```
gmapQ :: forall cxt, a. Data cxt a
      => (forall b. Data cxt b => b -> r)
      -> a -> [r]

gsize :: forall a. Size a => a -> Int
```

So in the call `(gmapQ gsize t)`, the function `f` can use any operations accessible from `Data cxt b`. In this case we want `cxt` to be `Size`, *but there is no way to say so*. The universally-quantified `cxt` type parameter in `gmapQ`'s type is mentioned *only in constraints*: it is ambiguous. However, if we could specify the type arguments to use, we would be fine:

```
-- Pseudo-code
```

```
instance Data Size t => Size t where
  gsize x = 1 + sum (gmapQ {|Size,t|} gsize x)
```

[3] This solution suffers from a difficulty discussed and solved in Section 5, but we pass lightly on since this is a failed attempt anyway.

Here, we imagine another non-standard extension to Haskell, namely the ability to specify the types at which a polymorphic function is called. The notation `gmapQ {|Size,t|}` means "`gmapQ` called with `cxt = Size` and `a = t`" (refer to the type of `gmapQ` given immediately above). We pass two type arguments, because `gmapQ` is quantified over two type parameters, but only the first is really interesting. Again, we will discuss how to encode this extension in standard Haskell, in Section 4.2, but the essential intent is simply to fix the type arguments for `gmapQ`.

3.3 The `Data` instances

As in our earlier work, every data type must be made an instance of class `Data`, either manually or with compiler support. For example, here are the instance declarations for integers and lists:

```
instance (cxt Int) => Data cxt Int where
  gmapQ f n = []

instance (cxt [a], Data cxt a)
         => Data cxt [a] where
  gmapQ f []     = []
  gmapQ f (x:xs) = [f x, f xs]
```

Compared to our earlier work, the only change is an extra context for each instance declaration — `(cxt Int)` and `(cxt [a])` respectively — to provide the necessary superclass. Here, we need an instance declaration context that contains structured types (e.g.,

`(cxt [a])`), so one might worry about the termination of constraint solving, a point we return to in Section 5.

3.4 Using the new customisation

In the type-class framework, new instances can be added (by the client of the `gsize` library) in an extremely straightforward manner. For example:

```
instance Size Name where
  gsize n = 1

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

The first instance declaration says that a `Name` always has size 1, regardless of the size of the `String` inside it (c.f. Section 2.1). The second instance defines the size of a list to be the sum of the sizes of its components, without counting the cons cells themselves, the `[]` at the end. (Both would be counted by the generic definition.)

One can make new generic functions by combining existing ones, just as you always can with type classes. For example, suppose we have a generic depth-finding function `gdepth`, defined similarly to `gsize`. Then we can combine them to find the "density" of a data structure:

```
density :: (Size a, Depth a) => a -> Int
density t = gsize t / gdepth t
```

Notice that the context is explicit about all the generic functions that are called in the body. Again, this is just standard type-class behaviour, and we could easily have a single class combining both `gsize` and `gdepth`.

3.5 Overlapping instances and default methods

So far we have given the *generic* definition of `gsize` – the one to use if not overridden – using an *instance* declaration thus:

```
instance Data Size t => Size t where
  gsize x = 1 + sum (gmapQ {|Size,t|} gsize x)
```

Notice the "`=> Size t`", which makes this instance overlap with every other instance of `Size`. Hence, this approach relies on overlapping instances, a non-Haskell 98 feature.

We can avoid overlapping instances, using Haskell 98's *default method* declarations instead. We briefly review default methods, using a trivial example:

```
class Num a where
  (+), (-) :: a -> a -> a
  negate :: a -> a
  (-) x y = x + negate y
```

The definition of `(-)` in the class declaration is the default method for `(-)`; if an instance declaration defines only `(+)` and `negate`, the method for `(-)` is filled in from the default method in the class declaration. A default method has the same "use this unless overridden" flavour as do our generic functions.

Consider our class `Size`:

```
class Size a where
  gsize :: a -> Int
  gsize x = ????
```

The default method for `gsize` can assume absolutely nothing about the type `a`, so it is hard for it to do anything useful. The obvious way to fix this is to add `Data` as a superclass of `Size`, thus:

```
class Data Size a => Size a where
```

```
gsize :: a -> Int
gsize x = 1 + sum (gmapQ {|Size,a|} gsize x)
```

Now, for every type for which we want to use the generic definition, we must add a boilerplate instance declaration. For instance:

```
instance Size a => Size [a]
instance (Size a, Size b) => Size (a,b)
```

These instances omit the code for `gsize`, so that it is filled in by the default-method code from the class declaration. Type-specific instances, such as that for `Size Name`, are written just as before, with explicit type-specific code for `gsize`.

Compared to the previous approach, using the default method has the the advantage that it does not require overlapping instances. There seem to be two disadvantages. First, since `Data` is now a superclass of `Size`, every type that is an instance of `Size` must also be an instance of `Data`, even though the methods of `Data` may be entirely unused for that type; this seems inelegant. Second, one must give an explicit (albeit brief) `Size` instance declaration for every type for which `gsize` is to be callable, including ones for which the generic behaviour is wanted (e.g., lists and pairs above). However, in some applications this "disadvantage" might be considered an advantage, because it forces the library client to make a conscious decision about whether to use a type-specific implementation for `gsize` (by supplying code in the instance declaration), or to use the generic method (by omitting the code).

3.6 Intermediate summary

We have now concluded our overview of the key new idea in this paper: if we can abstract over type classes, then we can arrange for modular customisation of generic functions, the challenge we posed in Section 2. Apart from modular extensibility, the approach

has several other benefits, compared to the cast-based technique of our earlier work:

- There are no run-time type tests. Instead, execution proceeds using the existing Haskell type-class mechanism: the overloading is resolved either statically, or by dictionary passing.
- There is no danger of accidentally extending a generic function in incompatible ways for the same data type. Any attempt to do so will be reported as an overlapping-instance error.
- No extra complexity is associated with customising the generic function at type constructors — for example, see the instance for `Size` on pairs in the previous sub-section. By contrast, in our earlier work [LP04], it required distinct generic function combinators for each new kind of type constructor.

We have assumed a number of extensions to Haskell 98:

- Multi-parameter type classes, a very well-established extension.
- Overlapping instance declarations are required in one formulation, but are entirely avoidable (Section 3.5).
- The ability to abstract over type classes. This extension can be encoded in ordinary Haskell (Section 4.1).
- Explicit type application; again this is readily encoded in ordinary Haskell (Section 4.2).
- The ability to declare an instance for `(Size t)`, where `t` is a type variable; and the possibility of non-type-variable constraints in the context of an instance declaration. Both these extensions are used in the instance declaration for `(Size t)` in Section 3.2, for example. They are both illegal in Haskell 98, in order to guarantee decidability of type inference.

Of these, the last is the only extension that is both unavoidable

and not already widely available. Decidability of type inference is indeed threatened. In Section 5, we describe a corresponding Haskell extension that is based on building recursive dictionaries.

4. Encoding in Haskell

In this section we show how to encode the technique discussed in Section 3 in Haskell with common extensions.

4.1 Encoding abstraction over classes

The biggest apparent difficulty is the question of abstraction over type classes. John Hughes encountered a very similar problem six years ago, in the context of a language concept for algebraic data types with attached restrictions, and he described a way to encode abstraction over type classes without extending Haskell [Hug99]. We can adopt Hughes' techniques for our purposes.

We begin by defining, once and for all, a class `Sat`, with a single method, `dict`[4]:

```
class Sat a where dict :: a
```

This class becomes a superclass of `Data`, thus:

```
class (Typeable a, Sat (cxt a)) => Data cxt a where
  gmapQ :: (forall b. Data cxt b => b -> r)
        -> a -> [r]
```

Now, whenever a generic-library author defines a new class for a generic function, such as `Size`, she additionally defines a new *record type* `SizeD`, which corresponds to the *dictionary type* for the envisaged class. The fields of the record type correspond one-to-one to the methods of the class:

```
data SizeD a = Size { gsizeD :: a -> Int }
```

This type automatically gives us a record selector with the following parametrically polymorphic type:

```
gsizeD :: SizeD a -> a -> Int
```

It happens in this case that there is only one method, but the encoding works equally well when there are many. Along with the new record type, we also give a new instance declaration for `Sat`:

---

[4] Short for "Satisfies" and class "dictionary", respectively.

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

As you can see, both the record type and the instance declaration are trivially derived from the class declaration of `Size`. Now the library author can give the generic definition for `gsize`, via an instance declaration for `Size t`, just as in Section 3.2:

```
instance Data SizeD t => Size t where
  gsize t = 1 + sum (gmapQ {|SizeD,t|}
                           (gsizeD dict) t)
```

Here comes the crucial point: *the recursive call to* `gsize` *is made by calling* `(gsizeD dict)`, instead of `gsize`, because the function passed to `gmapQ` only has access to `Data SizeD t`, and hence to `Sat (SizeD t)`, but not to `Size t`. Accidentally calling `gsize` instead of `(gsizeD dict)` would yield a type error.

It is only when one wants to call `gsize` inside an argument passed to a rank-2 polymorphic SYB combinator (such as `gmapQ`) that one has to call `gsizeD dict`. Type-specific code never has to do this. For example, the instances given in Section 3.4 work unchanged; no encoding is needed:

```
instance Size Name where
```

```
  gsize n = 1

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

In practise this means that the encoding effort for type-class abstraction is limited to generic function libraries; clients of such libraries will not be concerned with the encoding.

4.2 Explicit type application

In Section 3.2 we found that we needed to specify the type arguments for a call to `gmapQ`, which we did using the notation `gmapQ {|Size,t|}`. There is a standard way to treat this difficulty in standard Haskell, by using a *type-proxy parameter*. Suppose that we give `gmapQ` the following type:

```
gmapQ :: forall cxt, a. Data cxt a => Proxy cxt
      -> (forall b. Data cxt b => b -> r)
      -> a -> [r]
```

The function `gmap` gets a new formal parameter, of type `Proxy cxt`, so that the type of an actual parameter will fix the type `cxt`. The type `Proxy` does not need to define any constructor, as it is used for carrying around type information only:

```
data Proxy (cxt :: * -> *)
```

The actual type-proxy parameter for the `Size` context is constructed as follows:

```
sizeProxy :: Proxy Size
sizeProxy = error "urk"
```

As a result, we can now call `(gmapQ sizeProxy)` to fix the `cxt`

type argument of gmapQ to be Size:

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

We define sizeProxy to be error "urk", to emphasise that it is only used as a *type* proxy; its *value* is never examined. The definitions of gmapQ, in instance declarations for Data simply ignore the type-proxy argument. For example (notice the underbars):

```
instance (Sat (cxt [a]), Data cxt a)
         => Data cxt [a] where
  gmapQ _ f []     = []
  gmapQ _ (x:xs) = [f x, f xs]
```

In defining the type Proxy above, we took advantage of two GHC extensions. First, we omitted all the constructors, since we never build a concrete value of this type. Second, the type parameter cxt of Proxy has kind (* -> *), which we indicated with a kind signature. If we wanted to stick to vanilla Haskell 98, we could instead write:

```
data Proxy cxt = P (cxt Int)
   -- Any type other than Int would also be fine
```

The constructor P will never be used, but it specifies the kind of cxt via its use in the component (cxt Int).

Although we describe type-proxy arguments as an encoding of "proper" type arguments, they are in some ways superior. In the hypothetical extension of Section 3.2, allowing type arguments, we had to pass two type arguments {|Size,t|}, even though only one was of interest. With type proxies we can identify exactly which type arguments must be passed. Furthermore, omitting an explicit type-proxy argument will lead to a somewhat-comprehensible error

message, whereas omitting a genuine type argument might lead to a less-comprehensible ambiguity error.

4.3 Intermediate summary

The encoding we describe is not heavy. The `Sat` class and `Proxy` types are defined in the SYB library, along with `Data`, `Typeable` and much else; and the derivation of `Data` and `Typeable` instances is automated in GHC[5]. In addition to defining a class for the generic function, the author of a generic library must also define a corresponding (a) record type, (b) `Sat` instance, and (c) type proxy. These definitions are pure boilerplate, and take only a line or two each. One could employ Template Haskell [SP02] to eliminate the need to define (a)–(c) explicitly.

The only tricky points arise in writing the generic code for the function: the provision of type-proxy parameters, and the necessity of calling `(gizeD dict)` instead of `gsize` in SYB combinator arguments. The client of a generic library sees no encoding whatsoever. However, like any encoding, type errors are likely to be less perspicuous than if type-class abstraction were directly supported.

For completeness, Figure 1 gives a small but complete example, which executes in GHC. It is partitioned into the code that has to be written by the three protagonists.

4.4 Related work

Hughes encountered the need for abstraction over type classes in the context of restricting type parameters of abstract data type constructors [Hug99]. For instance, an operation for a membership test could be of potentially different types, depending on the actual data type constructors:

```
-- An Eq constraint would be fine
-- for a simple set data type
```

```
member :: Eq a => a -> PlainSet a -> Bool

-- An Ord constraint would be more efficient
-- for binary trees
member :: Ord a => a -> BinTree a -> Bool
```

Hence, the type could not be defined once and for all in a type class. Hughes therefore proposed to enable restricted algebraic data types, where `PlainSet` and `BinTree` will be constrained, *and* these constraints are *implied* by any use of the restricted data types

---

[5] As of writing this paper, compiler support is limited to the previous form of `Data` instances, but the source distribution for this paper includes templates (in the sense of Template Haskell) for the new form of `Data` instances.

```
module Example where

import Data.Typeable

--------- SYB library code -----------
data Proxy (a :: * -> *)

class Sat a where { dict :: a }

class (Typeable a, Sat (ctx a))
      => Data ctx a where
    gmapQ :: Proxy ctx
          -> (forall b. Data ctx b => b -> r)
          -> a -> [r]

instance Sat (cxt Char) => Data cxt Char where
  gmapQ _ f n = []

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

--------- gsize library code -----------
class Size a where gsize :: a -> Int

data SizeD a = SizeD { gsizeD :: a -> Int }

sizeProxy :: Proxy SizeD
sizeProxy = error "urk"

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)

--------- gsize client code -----------
instance Size a => Size [a] where
  gsize []     = 0
  gsize (x:xs) = gsize x + gsize xs

test = (gsize ['a', 'b'], gsize 'x')
            -- Result = (2,1)
```

Figure 1. Self-contained sample code for generic size

in type signatures or otherwise. Hughes proposed abstraction over type classes as an aid for the simulation of restricted data types. For instance, a collection class would be parameterised as follows:

```
class Collection c cxt where
  member :: cxt a => a -> c a -> Bool
```

Hughes made the point that restricted data types should receive extra language support, since the simulation based on "classes parameterised in classes" would require that the programmer anticipates extra parameters for constraints when designing classes such as `Collection`. In our case, the parametrisation in a superclass of `Data` is intuitive, which makes "classes parameterised in classes" an appropriate technique for SYB.

Hughes' *encoding* of abstraction over type classes comprised the `Sat` class, but the assumption was made that *existing* classes

should readily serve as parameters of other classes. In the SYB context, we need abstraction over type classes for the provision of *new* classes that implement generic functions. In fact, the default instance of such a new class (or the default method of the class) is the one and only client of the explicit dictionary.

5. Recursive dictionaries

Suppose we try to evaluate the expression `(gsize 'x')` for the program of Figure 1. The call to `gsize` gives rise to the constraint `Size Char`, which the type checker must discharge. Let us see how the constraint can be satisfied:

```
    Size Char
→   Data SizeD Char      Instance head Size t
→   Sat (SizeD Char)     Instance head Data cxt Char
→   Size Char            Instance head Sat (SizeD t)
                         ... etc. ...
```

To satisfy the constraint `Size Char` we select the generic instance with the head `Size t` (because there is no `Size` instance that is specific to `Char`). Using that instance declaration means that we must now satisfy `Data SizeD Char`. We use the instance declaration for `(Data cxt Char)`, also given in Figure 1, which in turn means that we must satisfy `Sat (SizeD Char)`. Using the instance declaration for `Sat (SizeD t)` means that we need to satisfy `Size Char` — *but this is the very constraint from which we started dictionary construction.* There is a danger that constraint solving will fail to terminate.

Indeed, the instance declaration for `(Data cxt Char)` is not legal Haskell 98:

```
instance Sat (cxt Char) => Data cxt Char where
  gmapQ _ f n = []
```

The instance is illegal because the context (before the "=>") does not consist of simple constraints; that is, constraints of the form `C` $\alpha_1 ... \alpha_n$, where the α_i are just type variables. Haskell 98 imposes this restriction on instance constraints precisely in order to ensure that constraint-solving always terminates. GHC requires the flag `-fallow-undecidable-instances` to accept the instance declaration, to highlight the danger of non-termination. (Hugs also supports such a flag.) Incidentally, this problem is not caused by the `Sat` encoding; it would arise, in the same way, if parameterisation over type classes were directly supported. (The problem arises even for a hard-coded superclass, as discussed in Section 3.1.)

5.1 Cycle-aware constraint resolution

For the present scenario, however, there is a simple solution to the non-termination problem: *build a recursive dictionary*. To this end, a Haskell type checker must detect and discharge cycles in constraint resolution. We will now specify and assess the approach taken in GHC.

We presume that constraint resolution is modelled by a function $solve(S, C)$ that solves a constraint C, by deducing it from a set of "given" constraints S. Recursive dictionaries require the following behaviour:

$$
\begin{array}{ll}
solve(S, C) & \\
\quad = & \textbf{succeed}, \text{ if } C \in S \\
\quad = & solve(S \cup C, (D_1, \ldots, D_n)) \\
& \quad \text{if there is a unique instance declaration} \\
& \quad \text{that can be instantiated to the form } (D_1, \ldots, D_n) \texttt{ => } C \\
\quad = & \textbf{fail}, \text{ otherwise}
\end{array}
$$

The key point is that in the recursive call to $solve$, we add C to the "given" constraints S before trying to solve the sub-problems

$(D_1, \ldots, D_n)$. Dictionary construction is merely an elaboration of this scheme for constraint resolution. In each step, the algorithm needs to construct a dictionary to witness the solution, and the effect of "adding C to S before the recursive call" is to build a recursive dictionary.

This technique does not guarantee that *solve* will terminate, of course. Consider the following declaration:

```
instance Foo [[a]] => Foo [a] where ...
```

Using this declaration to satisfy constraint `Foo [Char]`, say, simply yields a more complicated constraint `Foo [[Char]]`, and so on. Adding C to S before the recursive call does not solve the halting problem! It just makes *solve* terminate more often.

This technique does not guarantee either that the recursively dictionary is useful. Consider the following declaration:

```
instance Foo [a] => Foo [a] where ...
```

The type checker will terminate all right, but only by building a dictionary that is defined to be equal to itself; any attempt to use methods from the dictionary will loop at run-time. One might be able to impose useful restrictions on the form of instance heads so that well-founded recursion is enforced. This refinement is likely to require a global analysis of the program in question. We leave this as a topic for future work.

5.2 Related work

The general idea of adding a goal to the set of known facts before attempting to prove its sub-goals is, of course, far from new — it amounts to a co-inductive proof rather than an inductive one. In the programming-language area it crops up when one attempts to decide the subtyping relation on recursive types [Car86, BH97,

Pie02, LS04]. Our application is unusual in that we derive a recursive proof term from the co-inductive proof, namely a recursive definition of the dictionary we seek. Our approach also shares similarities with tabling and other attempts in logic programming that improve the termination behaviour of depth-first search and SLD resolution [SSW00].

Hughes's paper [Hug99] also mentioned the desirability of detecting loops in context reduction, but for a different reason, and with a different (and less satisfying solution). His problem concerned instance declarations that looked like

```
instance Sat (EqD a) => Eq a
instance Eq a => Sat (EqD a)
```

His proposal was that when an infinite loop like this was detected, the context-reduction search should back-track, and seek an alternative way to satisfy the constraints.

Our proposal is quite different. Looping context reductions succeed, and build a recursive dictionary, rather than failing as Hughes suggests. This extension to Haskell's context-reduction mechanism has been suggested several times. Here is a recent example. A programmer wanted to define and use the `Fix` data type:

```
data Fix f = In (f (Fix f))

data List a x = Nil | Cons a x

instance (Eq a, Eq x) => Eq (List a x) where
  Nil         == Nil         = True
  (Cons a b) == (Cons c d) = a == c && b == d
  other1      == other2      = False
```

Subject to an instance for `Fix`, we would like to test for equality of lists like the following:

```
test1, test2 :: Fix (List Char)
test1 = In Nil
test2 = In (Cons 'x' (In Nil))
```

The expression `(test1 == test2)` should evaluate to `False`! Equality on such lists ought to work because data structures are finite, and so are the types. But how can we give the equality instance for `Fix`? Here is the obvious attempt; the instance head paraphrases the data type declaration for `Fix`:

```
instance Eq (f (Fix f)) => Eq (Fix f) where
  (In a) == (In b)  =  a == b
```

Now, the expression `(test1 == test2)` gives rise to the constraint `Eq (Fix (List Char))`, whose simplification resembles unfolding steps of a recursive data type constructor:

```
    Eq (Fix (List Char))
→   Eq (List (Fix (List Char)))   Instance Eq (Fix f)
→   Eq (Fix (List Char))          Instance Eq (List a x)
→   Eq (List (Fix (List Char)))   ... etc. ...
```

In this case, too, building a recursive dictionary is precisely the right thing to do. Of *course* we need a recursive function, if we are to compute equality on a recursive type, and `Fix (List Char)` is indeed a recursive type, albeit indirectly.

6. Case study: QuickCheck

As a real-life illustration of the ideas of this paper, we now describe the `shrink` function from the QuickCheck library, referred to in the Introduction. For the sake of a concise notation, we will pretend that Haskell supports abstraction over classes, but everything in this section is readily encoded using Section 4; the actual code is in the source distribution that comes with the paper.

The Haskell library QuickCheck makes it easy to test functions. It generates random data of the appropriate type, feeds it to the function, and checks that the result satisfies a programmer-supplied criterion. QuickCheck is described by a fascinating series of papers [CH00, CH02b], but we concentrate here on a more recent development: its ability to refine failing cases. When QuickCheck finds inputs that make the function under test fail, these inputs are often not the *smallest* ones that make it fail. So it makes sense to successively "shrink" the failing input, until it no longer fails. This technique turns out to work surprisingly well in practise.

What is needed, then, is an overloaded function `shrink` that takes a value and returns a list of values of the same type, that have been shrunk by one "step":

```
class Shrink a where
  shrink :: a -> [a]

shrinkProxy :: Proxy Shrink
shrinkProxy = error "urk"
```

We return a list, because there is often more than one way to shrink a value, and there may be none (e.g., an integer cannot be shrunk). A "step" is the smallest shrinkage we can do to the value; by applying `shrink` many times, we can shrink a value by more than one step.

There are two obvious generic strategies for shrinking a value v:

1. Choose one of v's sub-components, where that sub-component is of the same type as v. For example, one way to shrink a list `(x:xs)` is to return just `xs`, because `xs` has the same type as `(x:xs)`.
2. Shrink one (and only one) of v's (immediate) sub-components by one step. For example, to shrink a pair `(a,b)` we can either

shrink a or shrink b.

These strategies suggest the following generic `Shrink` instance:

```
instance Data Shrink a => Shrink a where
  shrink t = children t ++ shrinkOne t
```

In the next two sections, we will write the helper functions `children` and `shrinkOne`. Meanwhile, whenever the user introduces a new data type `Foo`, she can either do nothing (and get the generic instance above), or give an explicit instance declaration to override it. The user may want to provide a data type-specific instance in order to ensure invariants during shrinking. For example:

```
data ListWithLength a = LWL [a] Int
   -- Invariant:
   --  the Int is the length of the list

instance Data Shrink a
      => Shrink (ListWithLength a) where
  shrink (LWL []     n) = []
  shrink (LWL (x:xs) n)
    = LWL xs (n-1) :
      [ LWL xs' n
      | xs' <- shrinkOne xs]
```

6.1 Finding compatibly-typed children

Let's write `children` first.
It is a generic function with the following type:

```
children :: Data Shrink a => a -> [a]
```

Its business is to look at each of the sub-components of its argument, and return the "largest" subcomponents that have the same type as the argument. (For simplicity, we will limit ourselves to *im-*

mediate subcomponents here.) The definition of `children` makes use of the type-safe cast operation:

```
children :: Data Shrink a => a -> [a]
children t
  = [c | Just c <- gmapQ shrinkProxy cast t]
```

Recall the type of `cast`:

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
```

where `Typeable` is a superclass of `Data`. The call `(cast x)` returns `Just x` if the context needs a value of the same type as `x` (that is, `a=b`), and `Nothing` otherwise. The generic map function, `gmapQ` applies `cast` to each of `t`'s immediate children, in a context that requires the result to have the type `Maybe` τ, where `t` has type τ, so all we need do is to collect the `Just` members of this list. Note that we pass `shrinkProxy` to `gmapQ`, even though `cast` does no shrinking; it needs only `Typeable`. We need to choose *some* proxy to comply with `gmapQ`'s type, and we have a `(Data Shrink a)` dictionary to hand.

6.2 Shrinking sub-components

Writing `shrinkOne` generically is a little harder. Recall that `(shrinkOne t)` should apply `shrink` to all the immediate subcomponents of `t`, and construct shrunken versions of `t` from the result. For example, suppose that:

```
shrinkOne x  =  [x1]
shrinkOne y  =  [y1,y2]
```

then the result of shrinking the pair `(x,y)` is this:

```
shrinkOne (x,y) = [(x1,y), (x,y1), (x,y2)]
```

Notice that each result has just one shrunken component. Before

thinking about implementing a *generic* `shrinkOne`, let us write a particular case. Here is `shrinkOne` for pairs:

```
shrinkOnePr :: (Shrink a, Shrink b)
            => (a,b) -> [(a,b)]
shrinkOnePr (x,y) =  [(x',y) | x' <- shrink x]
                  ++ [(x,y') | y' <- shrink y]
```

The more components, the more similar-looking list comprehensions we have to write. It would be nicer — and we are anticipating our needs for generic programming — to use do-notation:

```
shrinkOnePr :: (Shrink a, Shrink b)
            => (a,b) -> [(a,b)]
shrinkOnePr (x,y) =  do { x' <- shrink x
                        ; y' <- shrink y
                        ; return (x', y') }
```

This would not work, partly because the list monad forms *all combinations* of the results as opposed to combinations with one shrunk position only, and partly because there are *no combinations* that include the original `x` and `y`. We can solve both problems with a single blow, by using a different monad, like this:

```
data S a = S a [a]

shrinkS :: Shrink a => a -> S a
shrinkS t = S t (shrink t)
```

The idea is that `(shrinkS t)` returns a pair `(S t ts)`, containing the original argument `t` and a list of one-step shrunken versions of `t`. Then we give an `instance` declaration that makes `S` into a monad, in a different way to lists, with a more selective way of combining its components. For example:

```
do { x' <- S x [x1]
   ; y' <- S y [y1,y2]
   ; return (x',y') }
```

returns the list `[(x1,y), (x,y1), (x,y2)]`. Furthermore, the same pattern works no matter how many components are involved. Here is how we make `S` into a monad:

```
instance Monad S where
  return x = S x []

  (S x xs) >>= k
     = S r (rs1 ++ rs2)
     where
       S r rs1 = k x
       rs2 = [r | x <- xs, let S r _ = k x]
```

The case for `return` is easy. Now recall that `(>>=)` has type:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

The un-shrunk result `r` is obtained by passing the un-shrunk part `x` of the first argument `(S x xs)` to the rest of the computation `k`, and taking the un-shrunk part of the result. The shrunk parts of `(k x)`, namely `rs1` are useful too, because they are shrunk by one step, and so form part of the result. The other one-step shrunken results, `rs2`, are obtained by taking the shrunken parts `xs` of the first argument, passing them to the rest of the computation `k`, and taking the un-shrunken part of its result.

Now we can indeed write `shrinkOnePr` with do-notation, using `S` as its result type:

```
shrinkOnePr :: (Shrink a, Shrink b)
```

```
                => (a,b) -> S (a,b)
shrinkOnePr (x,y) =  do { x' <- shrinkS x
                        ; y' <- shrinkS y
                        ; return (x', y') }
```

All that remains is to do this generically. Since we want to combine results monadically, the combinator we need is the *monadic* map gmapM, a cousin of gmapQ [LP03]:

```
gmapM :: (Monad m, Data cxt a)
      => Proxy cxt
      -> (forall b. Data cxt b => b -> m b)
      -> a -> m a
```

Although gmapM is, in reality, defined using the yet-more-general combinator gfoldl, it can best be understood through its instances. For example, here is the instance for pairs:

```
instance (cxt (a,b), Data cxt a, Data cxt b)
       => Data cxt (a,b) where
   gmapM _ f (x,y)
     = do { x' <- f x
          ; y' <- f y
          ; return (x',y') }
```

Comparing this definition of gmapM for pairs with shrinkOnePr above, it should be clear that the generic code for shrinkOne is simply this:

```
shrinkOne :: Data Shrink a => a -> [a]
shrinkOne t = ts
  where
    S _ ts = gmapM shrinkProxy shrinkS t
```

6.3 Summary

This example shows nicely how important it is to have *extensible* generic functions. QuickCheck is a *library* and cannot, of course, anticipate all the data types that its clients will define. Furthermore, the clients must be able to override the generic definition of `shrink` at will, because the generic method of shrinking might break invariants of the data structure.

Shrinking is just one example of the need for extensible generic functions, but QuickCheck has many others. For example the overloaded function `arbitrary` supports the generation of random data; just like `shrink`, there is a sensible generic definition, but the client must be able to override it. Incidentally, our choice of `shrink` happens to illustrate the continuing usefulness of the type-safe `cast` function.

7. Discussion and variations

In this section we discuss various alternative design choices, and contrast the approach described here with our earlier work.

7.1 Run-time type tests

Does this paper render obsolete our earlier work on "scrap your boilerplate" [LP03, LP04], which relied on run-time type tests? No, it does not, for several reasons. First, run-time type tests remain extremely useful, as we saw in the `Shrink` example in Section 6. In Section 7.2, we will also employ `cast` to model twin traversal. Second, the extra clutter of the context parameters (in both types and terms) is a real disadvantage, especially when generic functions are used in a first-class way, as we will illustrate in Section 7.3.

Third, one sometimes positively *wants* to enumerate type-specific cases explicitly! This issue arises with Haskell's type classes today. Sometimes you have a list of (first-name, last-name) pairs: you might want to sort it lexicographically by last name,

then by first name. But the built-in `Ord` instance for pairs works the other way round, and Haskell gives no way to use different instances at different places in the program [WB89]. This often prompts programmers to define new data types, but that does not work when you want to sort a single type in more than one way. Indeed, Haskell's Prelude has a function `sortBy` that takes an explicit function to use as the ordering. In short, the whole approach of using instance declarations to incrementally extend functions (whether generic or not) is rather "global"; if you want more local behaviour then the classic SYB approach might be better.

Lastly, just as dynamic types support run-time composition of values that cannot be statically type-checked, so `extQ` and friends allow the special cases of a generic function to be composed dynamically.

7.2 Twin traversal

We may wonder about the generality of the new SYB style. Can we rephrase all classical generic programming examples as listed in [LP03, LP04] so that the generic functions are open rather than closed? There is one challenge: multi-parameter traversal — in particular, twin traversal as in generic equality. In old style, we had proposed the following definition of generic equality [LP04]:

```
geq :: Data a => a -> a -> Bool
geq x y = geq' x y
 where
  geq' :: forall a b. (Data a,Data b)
       => a -> b -> Bool
  geq' x y =      (toConstr x == toConstr y)
             && and (gzipWithQ geq' x y)
```

Here `gzipWithQ` is a generic variation on the standard `zipWith` operation. It zips two lists of immediate subterms ("kids"). When

recursing into kids with `gzipWithQ`, we use an independently polymorphic generic equality; c.f. "`forall a b`" in the type of `geq'`. Clearly, if we wanted to rephrase this approach directly to the new style "with class", we will naturally end up requiring type-class parameterisation for classes with *two* parameters. Alas, our parameterisation of `Data` is restricted to classes with a single type parameter.

Why do we need independent polymorphism? The recursion into kids, `(gzipWithQ geq' x y)` uses *curried* generic maps such that one generic map processes the kids of `x` to compute a list of partial applications of `geq'` that are used in an accumulator position when processing the kids of `y` with another generic (and accumulating) map. In order to model the list of partial applications as a normal *homogeneous* list, each partial application must be a generic function. (That is, we cannot record the types of the kids of `x` in the types of the partial applications.) This forced genericity of partial applications implies independent polymorphism.

Existential quantification combined with `cast` comes to our rescue. We can eliminate the heterogeneity of kid types, and thereby use a dependently polymorphic `geq` in recursive calls. This new technique works equally well for both old and new SYB style.

We start with an envelope that wraps castable values.
The only way to access such an envelope is indeed by casting:

```
data Pack = forall x. Typeable x => Pack x
unpack :: Typeable a => Pack -> Maybe a
unpack (Pack x) = cast x
```

Processing the kids of `x` and `y` is organised as an accumulating generic map over the kids of `y`, where the kids of `x` contribute the initial accumulator value in the form of a list of `Packed` kids.

```
geq :: Data a => a -> a -> Bool
```

```
geq x y
  = let ([], bools) = gmapAccumQ geq' x' y
     in and ((toConstr x == toConstr y) : bools)
 where
  x' = gmapQ Pack x
  geq' :: Data y => [Pack] -> y -> ([Pack],Bool)
  geq' (x:xs) y
    = (xs, maybe False (geq y) (unpack x))
```

Note that the single-type-parametric polymorphic operation `geq` is used for the recursive calls that compare pairs of kids. Hence, we can readily move `geq` to a generic-function class with a single type parameter. (This is not demonstrated here.) It is a nuisance that we need to perform `casts` for the kids of `x`. One can easily see that the `casts` will succeed for the case that `x` and `y` use the same outermost term constructor. Alas, the type system cannot infer this fact.

7.3 First class generic functions

An attractive feature of our earlier paper [LP03] is that generic functions are first class: in particular, they can be passed as arguments to other Haskell functions, and they can be returned as results. Our new scheme shares this advantage, but the additional static checks make it somewhat less convenient, as we discuss in the rest of this section.

A potent application of first-class status is the ability to modularise algorithms into tree traversal and node processing. For example, here is the definition of `everywhere`, taken from the original SYB paper:

```
  -- Old type
everywhere :: Data a
           => (forall b. Data b => b -> b)
```

```
          -> a -> a
```

The call `(everywhere f t)` takes a generic function `f` and a data structure `t`, and applies `f` to every node in `t`. Indeed, by writing a type synonym we can make the type even more perspicuous:

```
type GenericT = forall a. Data a => a -> a
everywhere :: GenericT -> GenericT
```

A generic transformer `GenericT` has type `a->a`, for any type `a` that is traversable (lies in class `Data`). The `everywhere` combinator takes a generic transformer that works on individual nodes, and returns a transformer that works on the entire tree.

Matters are not so easy now that `Data` has an extra parameter. To begin with, `everywhere`'s type must look more like this:

```
type GenericT cxt = forall b. Data cxt b => b -> b
everywhere :: GenericT cxt -> GenericT cxt
```

In addition, `everywhere` needs a proxy type parameter, just like `gmapQ` and its cousins (Section 4.2). So `everywhere`'s type is actually this one:

```
everywhere :: Proxy cxt
           -> GenericT cxt -> GenericT cxt
```

(For the record, we can eliminate *some* proxy arguments in nested compositions of generic functions by means of implicit parameters [LLMS00].) Now suppose we want to make an actual traversal `(everywhere pickyCtx pickyInc t)` where the node-processing function `pickyInc` is defined like this:

```
pickyInc :: ( Data IncEligible t
```

```
                , Data IncSalary t
                ) => t -> t
pickyInc t | incEligible t = incSalary t
           | otherwise     = t
```

The details of this restricted function for salary increase do not matter; what is important is that `pickyInc`'s context has *two* constraints. Alas, that makes it incompatible with `everywhere`, which passes exactly one dictionary to its argument (see `everywere`'s type above). Hence, there is no straightforward way to provide the needed type-proxy argument `pickyCtx`.

Assuming that Haskell provides proper abstraction over type classes, one option is to combine `IncEligible` and `IncSalary` into a single class, thus:

```
class (IncEligible a, IncSalary a) => PickyInc a
-- no methods needed
```

We instantiate this class as follows:

```
instance (IncEligible a, IncSalary a)
  => PickyInc a
```

Clearly, we prefer a (non-Haskell 98) generic instance here because we do not want to re-enumerate all types covered by `IncEligible` and `IncSalary`. The adapted definition of `pickyInc` is constrained by the new helper class:

```
pickyInc :: Data PickyInc t => t -> t
pickyInc = ... as before ...
```

One could imagine a more sophisticated form of abstraction over classes, that automates this clutter, so that a single `ctx` pa-

rameter may transport several constraints instead of just one. This is a topic for future work.

When we consider the *encoding* for abstraction over type classes, as defined in Section 4.1, we may avoid the definition of a helper class, but we must provide a composed dictionary type — a product of the dictionary types for `IncEligible` and `IncSalary`:

```
data PickyIncD a
   = PickyIncD { dictE :: IncEligibleD a,
                 dictI :: IncSalaryD a }
```

The corresponding `Sat` instance will simply construct a pair of dictionaries taking advantage of preexisting `Sat` instances for `IncEligibleD` and `IncSalaryD`. Now, to call `incSalary` we need to extract it from two layers of wrapping:

```
incSalary' :: Data PickyIncD a => a -> Int
incSalary' = incSalaryD (dictI dict)
```

This proliferation of different versions of the same generic function is tiresome.

8. More related work

The overall 'Scrap your boilerplate' approach has been compared to other work on generic programming in detail in [LP03, LP04]. Likewise, we discussed work related to type-class parameterisation and recursive dictionaries in the respective sections. Therefore, we are going to focus here on the new enhancement: open, generic functions.

8.1 Derivable type classes

The "derivable type classes" approach to generic programming [HP00] is closely related to the work we describe here. Both ap-

proaches assume that a generic function is defined as a method of a type class, and that the programmer writes type-specific cases simply by giving an ordinary instance declaration. The big difference is that in the derivable-type-class approach, the class definition specifies a kind of template that can be used to generate the boilerplate; for example:

```
class Size a where
 gsize :: a -> Int    -- Code not correct
 gsize {| Unit    |} Unit      = 1
 gsize {| a :*: b |} (a :*: b) = gsize a + gsize b
 gsize {| a :+: b |} (Inl a)   = gsize a
 gsize {| a :+: b |} (Inr b)   = gsize b

instance Size [a]
instance Size (a,b)
```

The definition of `gsize` in the `class` declaration is a kind of generalised default method. The argument in the funny brackets `{|..|}` is a type argument, and the function is defined by induction over the structure of the type. For each instance declaration that does not give an explicit method definition, such as the ones for lists and pairs here, the compiler generates a method from the template, by converting the instance type to a sum-of-products form, and passing it to the generic code. It can be tricky to get the generic code right; in this case, there is a bug in `gsize` because it counts `1` only for nullary constructors! Fixing this requires a second method in the class, which is quite annoying.

Derivable type classes require considerable compiler support. The mechanism we propose here requires much less, and what we do need is useful for other purposes.

8.2 Generic Haskell specifically

In more recent versions of Generic Haskell [CL03, LCJ03], generic function definitions can involve some sort of default cases. This allows the programmer to fabricate customised generic functions while reusing fully generic functions as a kind of default. This is a major improvement over earlier polytypic programming practise, where types-specific cases (if any) had to form an integral part of the polytypic function declaration. Generic Haskell's default cases properly support capture of recursion. That is, recursive occurrences of the reused generic function are properly redirected to the reusing (i.e., customised) generic function.

Our development shows that Haskell's existing type-class mechanism can be readily leveraged for open, generic functions with appropriate capture of recursion. Generic Haskell (including its support for customisation) requires very considerable compiler support, in fact, a new compiler.

8.3 Generics for the masses

Hinze's recent "generics for the masses" approach [Hin04] is similarly lightweight as 'Scrap your boilerplate'. The distinguishing feature of Hinze's new proposal is that it captures essential idioms of Generic Haskell in a Haskell 98-based model, which requires absolutely no extensions.

The "generics for the masses" approach exhibits an important limitation in the view of generic function customisation. That is, the *class* for generics would need to be adapted for each new type or type constructor that requires a specific case. This is a

pernicious form of non-modularity. Hinze has identified this issue and its consequences: the approach is not useful for a generic programming library.

8.4 Intensional type analysis

Intensional type analysis [HM95] has made major contributions to the notions typecase and induction on type structure. The earlier work favours structural type equivalence, where the notion of a nominal branch in a typecase does not occur. An exception is the recent lambda calculus $\lambda_{\mathcal{L}}$ [VWW05], where the typecase construct can involve branches for user-defined types. This calculus also addresses another limitation of early work on intensional type analysis: it allows one to compute the branches in a typecase expression by a join operation. So one can parameterise in branches. Parameterisation in classic SYB functions is similar in effect. Hence, $\lambda_{\mathcal{L}}$ does not yet support modular customisation. Likewise, all other techniques that aim at enabling type-safe cast as an operation in a functional language, e.g., [Wei00, BS02, CH02a], do not support modular customisation.

9. Conclusions and further work

We have used type-class abstraction and recursive type-class dictionaries to support open, generic functions in an enhanced 'Scrap your boilerplate' approach (aka SYB). This makes SYB useful for a new range of generic programming applications, namely ones that require generic functions that can later be customised as new data types are added. QuickCheck is a good example, but other include: provision of system-wide generic equality, read, show, and friends; serialisation libraries in the XML context; extensible language implementation frameworks; and refinement of generic functions even

on fixed data types.

The first SYB paper focused on traversal problems ("generic consumers") for complex data structures, such as those corresponding to data models or language syntaxes. The second paper added support for generic builders (the opposite of generic consumers or traversals), and it described a number of specific techniques such as type case for type constructors and multi-parameter traversal. The present, third paper complements cast-based customisation by type-class-based customisation, which we had until recently believed to be impossible.

There is plenty left to do. The new proposed extensions for recursive dictionaries and type-class abstraction deserve dedicated study of their own: termination conditions for generalised instance heads, well-foundedness conditions for the constructed recursive dictionaries, type system formalisation for type-class parameterisation and context composition. In addition, some correspondence results for different styles of generic programming need to be discovered. For instance, can we encode all of derivable type classes? Finally, new application domains of generic programming are ready to be explored: we have argued that first-class generic functions facilitate computation of generic functions. This calls for research on generic function memoisation and adaptive generic algorithms.

Acknowledgements

We thank Koen Claessen and Simon Foster for their thoughtful critique of our early SYB approaches leading to the insight that extensible generic functions are urgently needed. Many thanks to James Cheney, Simon Marlow and Fermin Reig for helpful feedback on drafts of this paper. We also acknowledge the opportunity to present this work at a Generic Haskell meeting in January 2005. Ulf Norell, Sean Seefried and Simon Foster contributed Template Haskell code for the derivation of instances of the `Data`

class. We are grateful for helpful comments by the ICFP referees.

References

[BH97] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In *Proc 3rd International Conference on Typed Lambda Calculi and Appliactions (TLCA'97), Nancy, France*, volume 1210 of *Lecture Notes in Computer Science*, pages 63–81. Springer Verlag, 1997.

[BS02] A.I. Baars and S.D. Swierstra. Typing dynamic typing. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2002)*, pages 157–166. ACM Press, 2002.

[Car86] L Cardelli. Amber. In G Cousineau, PL Curien, and B Robinet, editors, *Combinators and functional programming languages*. LNCS 242, Springer Verlag, 1986.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 268–279, Montreal, September 2000. ACM.

[CH02a] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 90–104. ACM Press, 2002.

[CH02b] Koen Claessen and John Hughes. Testing monadic code with QuickCheck. In Manuel Chakravarty, editor, *Proceedings of the 2002 Haskell Workshop, Pittsburgh*, October 2002.

[CL03] Dave Clarke and Andres Löh. Generic Haskell, Specifically. In *Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 21–47. Kluwer, B.V., 2003.

[Fos05] Simon D. Foster. "*HAIFA: The Haskell Application Interoperation Framework Architecture*"; web site, 2004–2005. `http://www.repton-world.org.uk/mediawiki/index.`

php/HAIFA_Wiki.

[Hin04] Ralf Hinze. Generics for the masses. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 236–243. ACM Press, 2004.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles Of Programming Languages (POPL 1995)*, pages 130–141. ACM Press, 1995.

[HP00] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 Haskell Workshop, Montreal*, number NOTTCS-TR-00-1 in Technical Reports, September 2000.

[Hug99] RJM Hughes. Restricted data types in haskell. In Erik Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28 in Technical Reports, 1999. ftp://ftp.cs.uu.nl/pub/RUU/CS/techreps/CS-1999/1999-28.pdf.

[LCJ03] Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*, pages 141–152. ACM Press, August 25–29 2003.

[LLMS00] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: dynamic scoping with static types. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL 2000)*, pages 108–118. ACM Press, 2000.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 26–37, New Orleans, January 2003. ACM.

[LP04] Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*,

pages 244–255, Snowbird, Utah, September 2004. ACM.

[LS04] K Zhuo Ming Lu and M Sulzmann. An implementation of subtyping among regular expression types. In *Proc Asian Programming Languages Symposium (APLAS'04)*, volume 3302 of *Lecture Notes in Computer Science*, pages 57–73. Springer Verlag, 2004.

[Pie02] Benjamin Pierce. *Types and programming languages*. MIT Press, 2002.

[SP02] T Sheard and SL Peyton Jones. Template meta-programming for Haskell. In Manuel Chakravarty, editor, *Proceedings of the 2002 Haskell Workshop, Pittsburgh*, October 2002.

[SSW00] Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. An abstract machine for efficiently computing queries to well-founded models. *J. Log. Program.*, 45(1-3):1–41, 2000.

[VWW05] Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An Open and Shut Typecase. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2005)*. ACM Press, January 2005.

[WB89] PL Wadler and S Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc 16th ACM Symposium on Principles of Programming Languages, Austin, Texas*. ACM, January 1989.

[Wei00] S. Weirich. Type-safe cast: (functional pearl). In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 58–67. ACM Press, 2000.