Scrap More Boilerplate: Reflection, Zips, and Generalised Casts

Ralf Lämmel
Vrije Universiteit & CWI, Amsterdam

Simon Peyton Jones
Microsoft Research, Cambridge

Abstract

Writing boilerplate code is a royal pain. Generic programming promises to alleviate this pain by allowing the programmer to write a generic "recipe" for boilerplate code, and use that recipe in many places. In earlier work we introduced the "Scrap your boilerplate" approach to generic programming, which exploits Haskell's existing type-class mechanism to support generic transformations and queries.

This paper completes the picture. We add a few extra "introspective" or "reflective" facilities, that together support a rich variety of serialisation and de-serialisation. We also show how to perform generic "zips", which at first appear to be somewhat tricky in our framework. Lastly, we generalise the ability to over-ride a generic function with a type-specific one.

All of this can be supported in Haskell with independently-useful extensions: higher-rank types and type-safe cast. The GHC implementation of Haskell readily derives the required type classes for user-defined data types.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.1.1 [**Programming Techniques**]: Functional Programming; D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms

Design, Languages

Keywords

Generic programming, reflection, zippers, type cast

1 Introduction

It is common to find that large slabs of a program consist of "boilerplate" code, which conceals by its bulk a smaller amount of "interesting" code. So-called *generic programming* techniques allow programmers to automate this "boilerplate", allowing effort to be focused on the interesting parts of the program.

In our earlier paper, "*Scrap your boilerplate*" [16], we described a new technique for generic programming, building on the type-class facilities in Haskell, together with two fairly modest extensions (Section 2). Our approach has several attractive properties: it allows the programmer to over-ride the generic algorithm at exactly the desired places; it supports arbitrary, mutually-recursive data types; it is an "open-world" approach, in which it is easy to add new data types; it works without inefficient conversion to some intermediate universal data type; and it does not require compile-time specialisation of boilerplate code.

The main application in our earlier paper was traversals and queries over rich data structures, such as syntax trees or terms that represent XML documents. However, that paper did not show how to implement some of the best-known applications of generic programming, such as printing and serialisation, reading and de-serialisation, and generic equality. These functions all require a certain sort of *type introspection*, or *reflection*.

In this paper we extend our earlier work, making the following new contributions:

- We show how to support a general form of type reflection, which allows us to define generic "show" and "read" functions as well as similar functions (Sections 3 and 4).
- These classical generic functions rely on a new reflection API, supported on a per-data-type basis (Section 5). Once defined, this API allows other generic reflective functions to be defined, such as test-data generators (Section 5.4).
- Functions like generic equality require us to "zip together" *two* data structures, rather than simply to traverse one. We describe how zipping can be accommodated in the existing framework (Section 6).
- A strength of the *Scrap your boilerplate* approach is that it it easy to extend a generic function to behave differently on particular, specified types. So far it has not been clear how to extend a generic function for particular type *constructors*. In Section 7 we explain why this ability is very useful, and show how to generalise our existing type-safe `cast` operator so that we can indeed express such generic function extension.

Everything we describe has been implemented in GHC, and many examples are available online at the boilerplate web site [17]. No new extensions to Haskell 98 are required, beyond the two already described in *Scrap your boilerplate*, namely (a) rank-2 types, and (b) type-safe cast. The latter is generalised, however, in Section 7.2.

Proc ACM International Conference on Functional Programming (ICFP'04), Snowbird, Utah, Sept 2004, pp244-255 (with typos fixed).

2 Background

To set the scene for this paper, we begin with a brief overview of the *Scrap your boilerplate* approach to generic programming. Suppose that we want to write a function that computes the size of an arbitrary data structure. The basic algorithm is "for each node, add the sizes of the children, and add 1 for the node itself". Here is the entire code for `gsize`:

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

The type for `gsize` says that it works over any type `a`, provided `a` is a *data* type — that is, that it is an instance of the class `Data`[1] The definition of `gsize` refers to the operation `gmapQ`, which is a method of the `Data` class:

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

(The class `Typeable` serves for nominal type cast as needed for the accommodation of type-specific cases in generic functions. We will discuss this class in Section 7, but it can be ignored for now.) The idea is that `(gmapQ f t)` applies the polymorphic function `f` to each of the immediate children of the data structure `t`. Each of these applications yields a result of type `r`, and `gmapQ` returns a list of all these results. Here are the concrete definitions of `gmapQ` at types `Maybe`, list, and `Int` respectively:

```
instance Data a => Data (Maybe a) where
  gmapQ f Nothing  = []
  gmapQ f (Just v) = [f v]

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

instance Data Int where
  gmapQ f i = []  -- An Int has no children!
```

Notice that `gmapQ` applies `f` only to the *immediate* children of its argument. In the second instance declaration above, `f` is applied to `x` and `xs`, resulting in a list of exactly two elements, regardless of how long the tail `xs` is. Notice too that, in this same declaration, `f` is applied to arguments of different types (`x` has a different type to `xs`), and that is why the argument to `gmapQ` must be a *polymorphic* function. So `gmapQ` must have a higher-rank type – that is, one with a `forall` to the left of a function arrow — an independently-useful extension to Haskell [20].

It should now be clear how `gsize` works for term `t` whose type is an instance of the class `Data`. The call `(gmapQ gsize t)` applies `gsize` to each of `t`'s immediate children, yielding a list of sizes. The standard function `sum :: [Int] -> Int` sums this list, and then we add 1.

The class `Data` plays a central role in this paper. Our earlier paper placed three generic mapping operations in class `Data`: the operation `gmapQ` for generic queries, as illustrated above, and the operations `gmapT` for transformations, and `gmapM` for monadic transformations. In fact, all such forms of mapping can be derived from a single operator `gfoldl` for generic folding, as we also described in the earlier paper. The instances of `Data` are easy to define, as we saw for the operation `gmapQ` above. The definition of `gfoldl` is equally simple. In fact, the instances are *so* easy and regular that a compiler can do the job, and GHC indeed does so, when instructed by a so-called "`deriving`" clause. For example

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
              deriving( Eq, Typeable, Data )
```

[1] Note: in our earlier paper [16] the class now called "`Data`" was called "`Term`".

The "`deriving( Eq )`" part is standard Haskell 98, and instructs the compiler to generate an instance declaration for `instance Eq a => Eq (Tree a)`. GHC extends this by supporting `deriving` for the classes `Typeable` and `Data` as well.

While the operation `gfoldl` is sufficient for transformations and queries, it is not enough for other applications of generic programming, as we shall shortly see. Much of the rest of the paper fills out the `Data` class with a few further, carefully-chosen operations.

3 Generic "show" and friends

We will now consider generic functions that take any data value whatsoever, and render it in some way. For instance, a generic show operation is a generic function that renders terms as text, and hence it is of the following type:

```
gshow :: Data a => a -> String
```

That is, `gshow` is supposed to take any data value (i.e. any instance of class `Data`), and to display it as a string. The generic function `gshow` has many variants. For example, we might want to perform binary serialisation with `data2bits`, where we turn a datum into a string of `Zeros` and `Ones` (Sections 3.2 and 3.3). We might also want to translate a datum into a rose tree with `data2tree`, where the nodes store constructor names (Section 3.4).

```
data2bits    :: Data a => a -> [Bit]
data2tree    :: Data a => a -> Tree String
```

A generalisation of `data2tree` can perform type erasure for XML.

3.1 Data to text

We can almost do `gshow` already, because it is very like `gsize`[2]:

```
gshow t =  "("
        ++ concat (intersperse " " (gmapQ gshow t)
        ++ ")"
```

Of course, this function only outputs parentheses!

```
gshow [True,False] = "(() (() ()))"
```

We need to provide a way to get the name of the constructor used to build a data value. It is natural to make this into a new operation of the class `Data`:

```
class Typeable a => Data a where
  ...
  toConstr :: a -> Constr
```

Rather than delivering the constructor name as a string, `toConstr` returns a value of an abstract data type `Constr`, which offers the function `showConstr` (among others – Section 5):

```
showConstr :: Constr -> String
```

Given this extra function we can write a working version of `gshow`:

```
gshow :: Data a => a -> String
gshow t
  = "(" ++ showConstr (toConstr t)
        ++ concat (intersperse " " (gmapQ gshow t))
        ++ ")"
```

We have made use of an intermediate data type `Constr` so that, as well as supporting `showConstr`, we can also offer straightforward extensions such as fixity:

```
constrFixity :: Constr -> Fixity
```

The type `Fixity` encodes the fixity and precedence of the constructor, and we can use that to write a more sophisticated version of `gshow` that displays constructors in infix position, with minimum parenthesisation.

[2] The standard function `concat :: [[a]] -> [a]` concatenates the elements of a list of lists, while `intersperse :: a -> [a] -> [a]` inserts its first argument between each pair of elements in its second argument.

Built-in data types, such as `Int`, are also instances of the `Data` class, so `(toConstr (3::Int))` is a value of type `Constr`. Applying `showConstr` to this value yields the string representation of the integer value `3`.

3.2 Binary serialisation

Our next application is binary serialisation, in which we want to encode a data value as a bit-string of minimum length:

```
data Bit = Zero | One
data2bits :: Data a => a -> [Bit]
```

Rather than outputting the constructor name as a wasteful string, the obvious thing to do is to output a binary representation of its *constructor index*, so we need another function over `Constr`:

```
constrIndex :: Constr -> ConIndex
type ConIndex = Int -- Starts at 1; 0 for undefined
```

But how many bits should be output, to distinguish the constructor from other constructors of the same data type? To answer this question requires information about the entire data type, and hence a new function, `dataTypeOf`:

```
class Typeable a => Data a where
  ...
  toConstr   :: a -> Constr
  dataTypeOf :: a -> DataType
```

We note that `dataTypeOf` never ever examines its argument; it only uses its argument as a proxy to look-up information about its data type.[3] The abstract data type `DataType` offers the operation `maxConstrIndex` (among others):

```
maxConstrIndex :: DataType -> ConIndex
```

Using these functions, we are in a position to write `data2bits`:

```
data2bits :: Data a => a -> [Bit]
data2bits t = encodeCon (dataTypeOf t) (toConstr t)
              ++ concat (gmapQ data2bits t)

-- The encoder for constructors
encodeCon :: DataType -> Constr -> [Bit]
encodeCon ty con = natToBin (max-1) (idx-1)
                where
                  max = maxConstrIndex ty
                  idx = constrIndex con
```

Here we have assumed a simple encoder for natural numbers `natToBin :: Int -> Int -> [Bit]` where `(natToBin m x)` returns a binary representation of `x` in the narrowest field that can represent `m`.

3.3 Fancy serialisation

One could easily imagine more sophisticated serialisers for data values. For example, one might want to use adaptive arithmetic coding to reduce the number of bits required for common constructors [23, 18]. To do this requires the serialiser to carry along the *encoder state*, and to update this state whenever emitting a new constructor. So the fancy encoder will have this signature, which simply adds a state to `encodeCon`'s signature:

```
data State   -- Abstract
initState :: State
encodeCon :: DataType -> Constr
          -> State -> (State, [Bit])
```

Now we just need to modify the plumbing in `data2bits`. At first blush, doing so looks tricky, because `gmapQ` knows nothing about passing a state, but we can use a standard trick by making `gmapQ` return a list of functions of type `[State -> (State,[Bit])]`:

```
data2bits :: Data a => a -> [Bit]
data2bits t = snd (show_bin t initState)

show_bin :: Data a => a -> State -> (State, [Bit])
show_bin t st = (st2, con_bits ++ args_bits)
 where
  (st1, con_bits)  = encodeCon (dataTypeOf t)
                               (toConstr t) st
  (st2, args_bits) = foldr do_arg (st1,[])
                           enc_args

  enc_args :: [State -> (State,[Bit])]
  enc_args = gmapQ show_bin t

  do_arg fn (st,bits) = (st', bits' ++ bits)
    where
      (st', bits') = fn st
```

Notice that the call to `gmapQ` partially applies `show_bin` to the children of the constructor, returning a list of state transformers. These are composed together by the `foldr do_arg`. Of course, the appending of bit-strings is not efficient, but that is easily avoided by using any O(1)-append representation of bit-strings (see e.g. [9]).

A more elegant approach would instead present the encoder in a monadic way:

```
data EncM a   -- The encoder monad
instance Monad EncM where ...
runEnc  :: EncM () -> [Bit]
emitCon :: DataType -> Constr -> EncM ()
```

The monad `EncM` carries (a) the sequence of bits produced so far and (b) any accumulating state required by the encoding technology, such as `State` above. The function `emitCon` adds a suitable encoding of the constructor to the accumulating output, and updates the state. The function `runEnc` runs its argument computation starting with a suitable initial state, and returns the accumulated output at the end. All the plumbing is now abstracted, leaving a rather compact definition:

```
data2bits :: Data a => a -> [Bit]
data2bits t = runEnc (emit t)

emit :: Data a => a -> EncM ()
emit t = do { emitCon (dataTypeOf t) (toConstr t)
            ; sequence_ (gmapQ emit t) }
```

Here, the standard monad function

```
sequence_ :: Monad m => [m a] -> m ()
```

is used to compose the list computations produced by `gmapQ emit`.

3.4 Type erasure

The rendering operations so far are all forms of serialisation. We can also render terms as *trees*, where we preserve the overall shape of the terms, but erase the heterogeneous types. For instance, we can easily turn a datum into a rose tree of the following kind:

```
data Tree a = Tree a [Tree a]
```

The rendering operation is easily defined as follows:

```
data2tree :: Data a => a -> Tree String
data2tree x = Tree (showConstr (toConstr x))
                   (gmapQ data2tree x)
```

Rendering data values as rose trees is the essence of type erasure for XML. Dually, producing data values from rose trees is the essence of type validation for XML. Generic functions for XML type erasure and type validation would necessarily reflect various technicalities of an XML binding for Haskell [21, 2]. So we omit the tedious XML-line of scenarios here.

[3] One could instead use a 'phantom type' for proxies, to make explicit that `dataTypeOf` does not care about values of type `a`, i.e.:

```
data Proxy a = Proxy
dataTypeOf ::  Proxy a —> DataType
```

4 Generic “read” and friends

Our rendering functions are all generic *consumers*: they consume a data structure and produce a fixed type (`String` or `[Bit]`). (Generic traversals that query a term, are also consumers.) The inverse task, of parsing or de-serialisation, requires generic *producers*, that consume a fixed type and produce a data structure. It is far from obvious how to achieve this goal.

The nub of the problem is this. We are sure to need a new member of the `Data` class, `fromConstr`, that is a kind of inverse of `toConstr`. But what is its type? The obvious thing to try is to reverse the argument and result of `toConstr`:

```
class Typeable a => Data a where
 ...
 toConstr   :: a -> Constr
 fromConstr :: Constr -> a  -- NB: not yet correct!
```

But simply knowing the *constructor* alone does not give enough information to build a value: we need to know what the children of the constructor are, too. But we can't pass the children as arguments to `fromConstr`, because then the type of `fromConstr` would vary, just as constructor types vary.

We note that the type `Constr -> a` could be used *as is*, if `fromConstr` returned a term constructor filled by bottoms (“⊥”). A subsequent application of `gmapT` could still fill in the sub-terms properly. However, this is something of a hack. Firstly, the bottoms imply dependence on laziness. Secondly, the approach fails completely for strict data types. So we seek another solution.

The solution we adopt is to pass a generic function to `fromConstr` that generates the children. To this end, we employ a monad to provide input for generation of children:

```
fromConstrM :: (Monad m, Data a)
            => (forall b. Data b => m b)
            -> Constr -> m a
```

We will first demonstrate `fromConstrM`, and then define it.

4.1 Text to data

Here is the code for a generic read, where we ignore the need to consume spaces and match parentheses:

```
gread :: Data a => String -> Maybe a
gread input = runDec input readM

readM :: Data a => DecM a
readM =
  do { constr <- parseConstr ??? -- to be completed
     ; fromConstrM readM constr }
```

The two lines of `readM` carry out the following steps:

1. Parse a `Constr` from the front of the input. This time we employ a parser monad, `DecM`, with the following signature:

   ```
   data DecM a   -- The decoder monad
   instance Monad DecM where ...
   runDec      :: String -> DecM a -> a
   parseConstr :: DataType -> DecM Constr
   ```

   The monad carries (a) the as-yet-unconsumed input, and (b) any state needed by the decoding technology. The function `parseConstr` parses a constructor from the front of the input, updates the state, and returns the parsed constructor. It needs the `DataType` argument so that it knows how many bits to parse, or what the valid constructor names are. (This argument still needs to be filled in for “???” above.)

2. Use `fromConstrM` to call `readM` successively to parse each child of the constructor, and construct the results into a value built with the constructor identified in step 1.

The function `runDec` runs the decoder on a particular input, discarding the final state and unconsumed input, and returning the result. In case the monadic presentation seems rather abstract, we briefly sketch one possible implementation of the `DecM` monad. A parser of type `DecM a` is represented by a function that takes a string and returns a depleted string together with the parsed value, wrapped in a `Maybe` to express the possibility of a parse error:

```
newtype DecM a = D (String -> Maybe (String, a))
```

The type `DecM` can be made an instance of `Monad` in the standard way (see [10], for example). It remains to define the parser for constructors. We employ a new function, `dataTypeConstrs`, that returns a list of all the constructors of a data type. We try to match each constructor with the beginning of the input, where we ignore the issue of constructors with overlapping prefixes:

```
parseConstr :: DataType -> DecM Constr
parseConstr ty = D (\s -> match s (dataTypeConstrs ty))
 where
  match :: String -> [Constr] -> Maybe (String, Constr)
  match _ [] = Nothing
  match input (con:cons)
    | take (length s) input == s
       = Just (drop (length s) input, con)
    | otherwise
       = match input cons
    where
      s = showConstr con
```

The same code for `gread`, with a different implementation of `DecM` and a different type for `runDec`, would serve equally well to read the binary structures produced by `data2bits`.

4.2 Defining `fromConstrM`

The function `fromConstrM` can be easily defined as a new member of the `Data` class, with the type given above. Its instances are extremely simple; for example:

```
instance Data a => Data [a] where
  fromConstrM f con
    = case constrIndex con of
        1 -> return []
        2 -> do { a <- f; as <- f; return (a:as) }
```

However, just as `gmapQ`, `gmapT` and `gmapM` are all instances of the highly parametric `gfoldl` operation, so we can define `fromConstrM` as an instance of the dual of `gfoldl` — a highly parametric operation for unfolding. This operation, `gunfold` needs to be added to the `Data` class:

```
class Typeable a => Data a where
  ...
  gunfold :: (forall b r. Data b
                      => c (b -> r) -> c r)
          -> (forall r. r -> c r)
          -> Constr
          -> c a
```

The two polymorphically typed arguments serve for building non-empty vs. empty constructor applications. In this manner, `gunfold` really dualises `gfoldl`, which takes two similar arguments for the traversal of constructor applications. The operations `gunfold` and `gfoldl` also share the use of a type constructor parameter `c` in their result types, which is key to their highly parametric quality.

The instances of `gunfold` are even simpler than those for `fromConstrM`, as we shall see in Section 5.1. The operation `fromConstrM` is easily derived as follows:

```
fromConstrM f = gunfold k z
 where
  k c = do { c' <- c; b <- f; return (c' b) }
  z = return
```

Here, the argument `z` in `(gunfold k z)` turns the empty constructor application into a monadic computation, while `k` unfolds one child, and combines it with the rest.

4.3 Getting hold of the data type

In the generic parser we have thus-far shown, we left open the question of how to get the `DataType` corresponding to the result type, to pass to `parseConstr`, the "`???`" in `readM`. The difficulty is that `dataTypeOf` needs an argument of the result type, but we have not yet built the result value.

This problem is easily solved, by a technique that we frequently encounter in type-class-based generic programming. Here is the code for `readM` without "`???`":

```
readM :: Data a => DecM a
readM = read_help
  where
    read_help
      = do { let ty  = dataTypeOf (unDec read_help)
           ; constr <- parseConstr ty
           ; fromConstrM readM constr }

unDec :: DecM a -> a
unDec = undefined
```

Here, `unDec`'s type signature maps the type `DecM a` to `a` as desired. Notice the recursion here, where `read_help` is used in its own right-hand side. But recall that `dataTypeOf` is not interested in the *value* of its argument, but only in its *type*; the lazy argument `(unDec read_help)` simply explains to the compiler what `Data` dictionary to pass to `dataTypeOf`.

Rather than using an auxiliary `unDec` function, there is a more direct way to express the type of `dataTypeOf`'s argument. That is, we can use lexically-scoped type variables, which is an independently useful Haskell extension. We rewrite `readM` as follows:

```
readM :: Data a => DecM a
readM = read_help
  where
    read_help :: DecM a
      = do { let ty = dataTypeOf (undefined::a)
           ; constr <- parseConstr ty
           ; fromConstrM readM constr }
```

The definition

```
read_help :: DecM a = ...
```

states that `read_help` should have the (monomorphic) type `DecM a`, for some type `a`, and furthermore brings the type variable `a` into scope, with the same scope as `read_help` itself. The argument to `dataTypeOf`, namely `(undefined::a)`, is constrained to have the same type `a`, because the type variable `a` is in scope. A scoped type variable is only introduced by a type signature directly attached to a pattern (e.g., `read_help :: DecM a`). In contrast, a separate type signature, such as

```
read_help :: Data a => DecM a
```

is short for

```
read_help :: forall a. Data a => DecM a
```

and does not introduce any scoping of type variables. However, we stress that, although convenient, lexically-scoped type variables are not required to support the *Scrap your boilerplate* approach to generic programming, as we illustrated with the initial definition of `read_help`.

5 Type reflection — the full story

The previous two sections have introduced, in a piecemeal fashion, three new operations in the `Data` class. In this section we summarise these extensions. The three new operations are these:

```
class Typeable a => Data a where
  ...
  dataTypeOf :: a -> DataType
  toConstr   :: a -> Constr
  gunfold    :: (forall b r. Data b => c (b -> r) -> c r)
             -> (forall r. r -> c r)
             -> Constr
             -> c a
```

Every instance of `dataTypeOf` is expected to be non-strict — i.e. does not evaluate its argument. By contrast, `toConstr` must be strict — at least for multi-constructor types — since it gives a result that depends on the constructor with which the argument is built.

The function `dataTypeOf` offers a facility commonly known as "reflection". Given a type — or rather a lazy value that serves as a proxy for a type — it returns a data structure (`DataType`) that describes the structure of the type. The data types `DataType` and `Constr` are abstract:

```
data DataType -- Abstract, instance of Eq
data Constr   -- Abstract, instance of Eq
```

The following sections give the observers and constructors for `DataType` and `Constr`.

5.1 Algebraic data types

We will first consider algebraic data types, although the API is defined such that it readily covers primitive types as well, as we will explain in the next section. These are the observers for `DataType`:

```
dataTypeName    :: DataType -> String
dataTypeConstrs :: DataType -> [Constr]
maxConstrIndex  :: DataType -> ConIndex
indexConstr     :: DataType -> ConIndex -> Constr
type ConIndex   =  Int      -- Starts at 1
```

These functions should be suggestive, just by their names and types. For example, `indexConstr` takes a constructor index and a `DataType`, and returns the corresponding `Constr`. These are the observers for `Constr`:

```
constrType   :: Constr -> DataType
showConstr   :: Constr -> String
constrIndex  :: Constr -> ConIndex
constrFixity :: Constr -> Fixity
constrFields :: Constr -> [String]
data Fixity  =  ...    -- Details omitted
```

(The name of `showConstr` is chosen for its allusion to Haskell's well-known `show` function.) We have already mentioned all of these observers in earlier sections, except `constrType` which returns the constructor's `DataType`, and `constrFields` which returns the list of the constructor's field labels (or `[]` if it has none). Values of types `DataType` and `Constr` are constructed as follows:

```
mkDataType :: String -> [Constr] -> DataType
mkConstr   :: DataType -> String -> [String]
           -> Fixity -> Constr
```

The function `readConstr` parses a given string into a constructor; it returns `Nothing` if the string does not refer to a valid constructor:

```
readConstr :: DataType -> String -> Maybe Constr
```

When the programmer defines a new data type, and wants to use it in generic programs, it must be made an instance of `Data`. GHC will derive these instance if a `deriving` clause is used, but there is no magic here — the instances are easy to define manually if desired. For example, here is the instance for `Maybe`:

```
instance Data a => Data (Maybe a) where
  ... -- gfoldl as before
  dataTypeOf _       = maybeType
  toConstr (Just _)  = justCon
  toConstr Nothing   = nothingCon
```

```
  gunfold k z con   =
    case constrIndex con of
      1 -> z Nothing  -- no children
      2 -> k (z Just) -- one child, hence one k

justCon, nothingCon :: Constr
nothingCon = mkConstr maybeType "Nothing" [] NoFixity
justCon    = mkConstr maybeType "Just"    [] NoFixity

maybeType :: DataType
maybeType =  mkDataType "Prelude.Maybe"
                        [nothingCon, justCon]
```

Notice that the constructors mention the data type and vice versa, so that starting from either one can get to the other. Furthermore, this mutual recursion allows `mkDataType` to perform the assignment of constructor indices: the fact that `Nothing` has index 1 is specified by its position in the list passed to `mkDataType`.

5.2 Primitive types

Some of Haskell's built-in types need special treatment. Many built-in types are explicitly specified by the language to be algebraic data types, and these cause no problem. For example, the boolean type is specified like this:

```
data Bool = False | True
```

There are a few types, however, *primitive types*, that cannot be described in this way: `Int`, `Integer`, `Float`, `Double`, and `Char`. (GHC happens to *implement* some of these as algebraic data types, some with unboxed components, but that should not be revealed to the programmer.) Furthermore, GHC adds several others, such as `Word8`, `Word16`, and so on.

How should the "reflection" functions, `dataTypeOf`, `toConstr`, and so on, behave on primitive types? One possibility would be to support `dataTypeOf` for primitive types, but not `toConstr` and `fromConstr`. That has the disadvantage that every generic function would need to define special cases for all primitive types. While there are only a fixed number of such types, it would still be tiresome, so we offer a little additional support.

We elaborate `Constr` so that it can represent a value of primitive types. Then, `toConstr` constructs such specific representations. While `Constr` is opaque, we provide an observer `constrRep` to get access to constructor representations:

```
constrRep :: Constr -> ConstrRep
data ConstrRep
  = AlgConstr  ConIndex -- Algebraic data type
  | IntConstr   Integer -- Primitive type (ints)
  | FloatConstr  Double -- Primitive type (floats)
  | StringConstr String -- Primitive type (strings)
```

The constructors from an algebraic data type have an `AlgConstr` representation, whose `ConIndex` distinguishes the constructors of the type. A `Constr` resulting from an `Int` or `Integer` value will have an `IntConstr` representation, e.g.:

```
constrRep (toConstr (1::Int))  ==  IntConstr 1
```

The same `IntConstr` representation is used for GHC's data types `Word8`, `Int8`, `Word16`, `Int16`, and others. The `FloatConstr` representation is used for `Float` and `Double`, while `StringConstr` is used for anything else that does not fit one of these more efficient representations. We note that `Char`s are represented as `Integer`s, and `String`s are represented as lists of `Integer`s.

There is a parallel refinement of `DataType`:

```
dataTypeRep :: DataType -> DataRep
data DataRep
  = AlgRep [Constr]     -- Algebraic data type
  | IntRep              -- Primitive type (ints)
  | FloatRep            -- Primitive type (floats)
  | StringRep           -- Primitive type (strings)
```

There are dedicated constructors as well:

```
mkIntType      :: String -> DataType
mkFloatType    :: String -> DataType
mkStringType   :: String -> DataType
mkIntConstr    :: DataType -> Integer -> Constr
mkFloatConstr  :: DataType -> Double  -> Constr
mkStringConstr :: DataType -> String  -> Constr
```

The observers `constrType`, `showConstr` and `readConstr` all work for primitive-type `Constr`s. All that said, the `Data` instance for a primitive type, such as `Int`, looks like this:

```
instance Data Int where
  gfoldl k z c = z c
  gunfold k z c = case constrRep c of
                   IntConstr x -> z (fromIntegral x)
                   _           -> error "gunfold"
  toConstr x = mkIntConstr intType (fromIntegral x)

intType = mkIntType "Prelude.Int"
```

5.3 Non-representable data types

Lastly, it is convenient to give `Data` instances even for types that are not strictly *data* types, such as function types or monadic `IO` types. Otherwise `deriving ( Data )` would fail for a data type that had even one constructor with a functional argument type, so the user would instead have to write the `Data` instance by hand. Instead, we make all such types into vacuous instances of `Data`. Traversal will safely cease for values of such types. However, values of these types can not be read and shown.

For example, the instance for `(->)` is defined as follows:

```
instance (Data a, Data b) => Data (a -> b) where
  gfoldl k z c  = z c
  gunfold _ _ _ = error "gunfold"
  toConstr _    = error "toConstr"
  dataTypeOf _  = mkNoRepType "Prelude.(->)"
```

Here we assume a trivial constructor for non-representable types:

```
mkNoRepType :: String -> DataType
```

To this end, the data type `DataRep` provides a dedicated alternative:

```
data DataRep = ...     -- As before
             | NoRep   -- Non-representable types
```

Some of GHC's extended repertoire of types, notably `Ptr`, fall into this group of non-representable types.

5.4 Application: test-data generation

As a further illustration of the usefulness of `dataTypeOf`, we present a simple generic function that enumerates the data structures of any user defined type. (The utility of generic programming for test-data generation has also been observed elsewhere [14].) Such test-data generation is useful for stress testing, differential testing, behavioural testing, and so on. For instance, we can use systematic test-data generation as a plug-in for QuickCheck [3].

Suppose we start with the following data types, which constitute the abstract syntax for a small language:

```
data Prog = Prog Dec Stat
data Dec  = Nodec | Ondec Id Type | Manydecs Dec Dec
data Id   = A | B
data Type = Int | Bool
data Stat = Noop | Assign Id Exp | Seq Stat Stat
data Exp  = Zero | Succ Exp
```

We want to define a generic function that generates all terms of a given finite depth. For instance:

```
> genUpTo 3 :: [Prog]
[Prog Nodec Noop, Prog Nodec (Assign A Zero),
 Prog Nodec (Assign B Zero), Prog Nodec (Seq Noop
```

```
Noop), Prog (Ondec A Int) Noop, Prog (Ondec A Int)
(Assign A Zero), Prog (Ondec A Int) (Assign B Zero),
Prog (Ondec A Int) (Seq Noop Noop), ... ]
```

Here is the code for `genUpTo`:

```
genUpTo :: Data a => Int -> [a]
genUpTo 0 = []
genUpTo d = result
 where
  -- Recurse per possible constructor
  result = concat (map recurse cons)

  -- Retrieve constructors of the requested type
  cons :: [Constr]
  cons = dataTypeConstrs (dataTypeOf (head result))

  -- Find all terms headed by a specific Constr
  recurse :: Data a => Constr -> [a]
  recurse = fromConstrM (genUpTo (d-1))
```

The non-trivial case ($d > 0$) begins by finding `cons`, the list of all the constructors of the result type. Then it maps `recurse` over `cons` to generate, for each `Constr`, the list of all terms of given depth with that constructor at the root. In turn, `recurse` works by using `fromConstrM` to run `genUpTo` for each child. Here we take advantage of the fact that Haskell's list type is a monad, to produce a result list that consists of all combinations of the lists returned by the recursive calls.

The reason that we bind `result` in the where-clause is so that we can mention it in the type-proxy argument to `dataTypeOf`, namely `(head result)` — see Section 4.3.

Notice that we have not taken account of the possibility of primitive types in the data type — indeed, `dataTypeConstrs` will fail if given a primitive `DataType`. There is a genuine question here: what value should we return for (say) an `Int` node? One very simple possibility is to return zero, and this is readily accommodated by using `dataRep` instead of `dataTypeConstrs`:

```
  cons = case dataTypeRep ty of
           AlgRep cons -> cons
           IntRep      -> [mkIntConstr ty 0]
           FloatRep    -> [mkIntConstr ty 0]
           StringRep   -> [mkStringConstr ty "foo"]
   where
    ty = dataTypeOf (head result)
```

We might also pass around a random-number generator to select primitive values from a finite list of candidates. We can also refine the illustrated approach to accommodate other coverage criteria [15]. We can also incorporate predicates into term generation so that only terms are built that meet some side conditions in the sense of attribute grammars [6]. Type reflection makes all manner of clever test-data generators possible.

6 Generic zippers

In our earlier paper, all our generic functions consumed a *single* data structure. Some generic functions, such as equality or comparison, consume *two* data structures at once. In this section we discuss how to program such zip-like functions. The overall idea is to define such functions as curried higher-order generic functions that consume position after position.

6.1 Curried queries

Consider first the standard functions `map` and `zipWith`:

```
map     ::     (b->c) ->        [b] -> [c]
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
```

By analogy, we can attempt to define `gzipWithQ` — a two-argument version of `gmapQ` thus. The types compare as follows:

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

gzipWithQ :: (Data a1, Data a2)
          => (forall b1 b2. (Data b1, Data b2)
                         => b1 -> b2 -> r)
          -> a1 -> a2 -> [r]
```

The original function, `(gmapQ f t)`, takes a polymorphic function `f` that it applies to each immediate child of `t`, and returns a list of the results. The new function, `(gzipWithQ f t1 t2)` takes a polymorphic function `f` that it applies to *corresponding pairs of* the immediate children of `t1` and `t2`, again returning a list of the results. For generality, we do not constrain `a1` and `a2` to have the same outermost type constructor, an issue to which we return in Section 6.5.

We can gain extra insight into these types by using some type abbreviations. We define the type synonym `GenericQ` as follows:

```
type GenericQ r = forall a. Data a => a -> r
```

That is, a value of type `GenericQ r` is a generic query function that takes a value of any type in class `Data` and returns a value of type `r`. Haskell 98 does not support type synonyms that contain `forall`'s, but GHC does as part of the higher-rank types extension. Such extended type synonyms are entirely optional: they make types more perspicuous, but play no fundamental role.

Now we can write the type of `gmapQ` as follows:

```
gmapQ :: GenericQ r -> GenericQ [r]
```

We have taken advantage of the type-isomorphism $\forall a.\sigma_1 \to \sigma_2 \equiv \sigma_1 \to \forall a.\sigma_2$ (where $a \notin \sigma_1$) to rewrite `gmapQ`'s type as follows:

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

Applying `GenericQ`, we obtain `GenericQ r -> GenericQ [r]`. So `gmapQ` thereby stands revealed as a *generic-query transformer*.

The type of `gzipWithQ` is even more interesting:

```
gzipWithQ :: GenericQ (GenericQ r)
          -> GenericQ (GenericQ [r])
```

The argument to `gzipWithQ` is a generic query that returns a generic query. This is ordinary currying: when the function is applied to the first data structure, it returns a function that should be applied to the second data structure. Then `gzipWithQ` is a transformer for such curried queries. Its implementation will be given in Section 6.3.

6.2 Generic comparison

Given `gzipWithQ`, it is easy to define a generic equality function:

```
geq' :: GenericQ (GenericQ Bool)
geq' x y =  toConstr x == toConstr y
         && and (gzipWithQ geq' x y)
```

That is, `geq' x y` checks that `x` and `y` are built with the same constructor and, if so, zips together the children of `x` and `y` with `geq'` to give a list of Booleans, and takes the conjunction of these results with `and :: [Bool] -> Bool`. That is the entire code for generic equality. Generic comparison (returning `LT`, `EQ`, or `GT`) is equally easy to define.

We have called the function `geq'`, rather than `geq`, because it has a type that is more polymorphic than we really want. If we spell out the `GenericQ` synonyms we obtain:

```
geq' :: (Data a1, Data a2) => a1 -> a2 -> Bool
```

But we do not expect to take equality between values of different types, `a1` and `a2`, even if both do lie in class `Data`! The real function we want is this:

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

Why can't we give this signature to the original definition of `geq'`? Because if we did, the call `(gzipWithQ geq' x y)` would be ill-typed, because `gzipWithQ` requires a function that is independently polymorphic in its two arguments. That, of course, just begs the question of whether `gzipWithQ` could be less polymorphic, to which we return in Section 6.5. First, however, we describe the implementation of `gzipWithQ`.

6.3 Implementing gzipWithQ

How can we implement `gzipWithQ`? At first it seems difficult, because we must simultaneously traverse two unknown data structures, but the `gmap` combinators are parametric in just one type. The solution lies in the type of `gzipWithQ`, however: *we seek a generic query that returns a generic query*. So we can evaluate `(gzipWithQ f t1 t2)` in two steps, thus:

```
gzipWithQ f t1 t2   -- NB: not yet correct!
  = gApplyQ (gmapQ f t1) t2
```

Step 1: use the ordinary `gmapQ` to apply `f` to all the children of `t1`, yielding a list of generic queries.

Step 2: use an operation `gApplyQ` to apply the queries in the produced list to the corresponding children of `t2`.

Each of these steps requires a little work. First, in step 1, what is the type of the list `(gmapQ f t1)`? It should be a list of generic queries, each of which is a *polymorphic* function. But GHC's support for higher-rank type still maintains *predicativity*. What this means is that while we can pass a polymorphic function as an argument, we cannot make a list of polymorphic functions. Since that really is what we want to do here, we can achieve the desired result by wrapping the queries in a data type, thus:

```
newtype GQ r = GQ (GenericQ r)

gzipWithQ f t1 t2
  = gApplyQ (gmapQ (\x -> GQ (f x)) t1) t2
```

Now the call to `gmapQ` has the result type `[GQ r]`, which is fine. The use of the constructor `GQ` serves as a hint to the type inference engine to perform generalisation at this point; there is no run-time cost to its use.

Step 2 is a little harder. A brutal approach would be to add `gApplyQ` directly to the class `Data`. As usual, the instances would be very simple, as we illustrate for lists:

```
class Typeable a => Data a where
  ...
  gApplyQ :: [GQ r] -> a -> [r]

instance Typeable a => Data [a] where
  ...
  gApplyQ [GQ q1, GQ q2] (x:xs) = [q1 x, q2 xs]
  gApplyQ []             []     = []
```

But we can't go *on* adding new functions to `Data`, and this one seems very specific to queries, so we might anticipate that there will be others yet to come.

Fortunately, `gApplyQ` can be defined in terms of the generic folding operation `gfoldl` from our original paper, as we now show. To implement `gApplyQ`, we want to perform a fold on immediate subterms while using an *accumulator* of type `([GQ r], [r])`. Again, for lists, the combination of such accumulation and folding or mapping is a common idiom (cf. `mapAccumL` in module `Data.List`). For each child we *consume* an element from the list of queries (component `[GQ r]`), while *producing* an element of the list of results (component `[r]`). So we want a combining function `k` like this:

```
k :: Data c => ([GQ r], [r]) -> c -> ([GQ r], [r])
k (GQ q : qs, rs) child = (qs, q child : rs)
```

Here `c` is the type of the child. The function `k` simply takes the accumulator, and a child, and produces a new accumulator. (The results accumulate in reverse order, but we can fix that up at the end using `reverse`, or we use the normal higher-order trick for accumulation.) We can perform this fold using `gfoldl`, or rather a trivial instance thereof — `gfoldlQ`:

```
gApplyQ :: Data a => [GQ r] -> a -> [r]
gApplyQ qs t = reverse (snd (gfoldlQ k z t))
  where
    k (GQ q : qs, rs) child = (qs, q child : rs)
    z = (qs, [])
```

The folding function, `gfoldlQ` has this type[4]:

```
gfoldlQ :: (r -> GenericQ r) -> r -> GenericQ r
```

The definition of `gfoldlQ` employs a type constructor `R` to mediate between the highly parametric type of `gfoldl` and the more specific type of `gfoldlQ`:

```
newtype R r x = R { unR :: r }
gfoldlQ k z t = unR (gfoldl k' z' t)
 where
  z' _ = R z -- replacement of constructor
  k' (R r) c = R (k r c) -- fold step for child c
```

6.4 Generic zipped transformations

We have focused our attention on generic zipped *queries*, but all the same ideas work for generic zipped *transformations*, both monadic and non-monadic. For example, we can proceed for the latter as follows. We introduce a type synonym, `GenericT`, to encapsulate the idea of a generic transformer:

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

Then `gmapT`, from our earlier paper, appears as a generic transformer transformer; its natural generalisation, `gzipWithT`, is a curried-transformer transformer:

```
gmapT     :: GenericT -> GenericT
gzipWithT :: GenericQ GenericT -> GenericQ GenericT
```

The type `GenericQ GenericT` is a curried two-argument generic transformation: it takes a data structure and returns a function that takes a data structure and returns a data structure. We leave its implementation as an exercise for the reader, along with similar code for `gzipWithM`. Programmers find these operations in the generics library [17] that comes with GHC.

6.5 Mis-matched types or constructors

At the end of Section 6.2, we raised the question of whether `gzipWithQ` could not have the less-polymorphic type:

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

Then we could define `geq` directly in terms of `gzipWithQ'`, rather than detouring via `geq'`. One difficulty is that `gzipWithQ'` is now not polymorphic *enough* for some purposes: for example, it would not allow us to zip together a list of booleans with a list of integers. But beyond that, an implementation of `gzipWithQ'` is problematic. Let us try to use the same definition as for `gzipWithQ`:

```
gzipWithQ' f t1 t2    -- Not right yet!
  = gApplyQ (gmapQ (\x -> GQ (f x)) t1) t2
```

The trouble is that `gApplyQ` requires a list of *polymorphic* queries as its argument, and for good reason: there is no way to ensure statically that each query in the list given to `gApplyQ` is applied to an argument that has the same type as the child from which the query was built. Alas, in `gzipWithQ'` the query `(f x)` is monomorphic,

[4]Exercise for the reader: define `gmapQ` using `gfoldlQ`. Hint: use the same technique as you use to define `map` in terms of `foldl`.

because `f`'s two arguments have the same type. However, we can turn the monomorphic query `(f x)` into a polymorphic one, albeit inelegantly, by using a dynamic type test: we simply replace the call `(f x)` by the following expression:

```
(error "gzipWithQ' failure" `extQ` f x)
```

The function `extQ` (described in our earlier paper, and reviewed here in Section 7.1) over-rides a polymorphic query (that always fails) with the monomorphic query `(f x)`.

Returning to the operation `gzipWithQ`, we can easily specialise `gzipWithQ` at more specific types, just as we specialised `geq'` to `geq`. For example, here is how to specialise it to list arguments:

```
gzipWithQL :: (Data a1, Data a2)
   => (forall b1,b2. (Data b1, Data b2) => b1 -> b2 -> r)
   -> [a1] -> [a2] -> [r]
gzipWithQL = gzipWithQ
```

A related question is this: what does `gzipWithQ` do when the constructors of the two structures do not match? Most of the time this question does not arise. For instance, in the generic equality function of Section 6.2 we ensured that the structures had the same constructor before zipping them together. But the `gzipWithQ` implementation of Section 6.3 is perfectly willing to zip together different constructors: it gives a pattern-match failure if the second argument has more children than the first, and ignores excess children of the second argument. We could also define `gzipWithQ` such that it gives a pattern-match failure if the two constructors differ. Either way, it is no big deal.

7 Generic function extension

One of the strengths of the *Scrap your boilerplate approach* to generic programming, is that it is very easy to extend, or over-ride, the behaviour of a generic function at particular types. To this end, we employ nominal type-safe cast, as opposed to more structural notions in other approaches. For example, recall the function `gshow` from Section 3:

```
gshow :: Data a => a -> String
```

When `gshow` is applied to a value of type `String` we would like to over-ride its default behaviour. For example, `(gshow "foo")` should return the string `"\"foo\""` rather than the string `"(: 'f' (: 'o' (: 'o' [])))"`, which is what `gshow` will give by default, since a `String` is just a list of characters.

The key idea is to provide a type-safe `cast` operation, whose realisation formed a key part of our earlier paper; we review it in Section 7.1. However, further experience with generic programming reveals two distinct shortcomings, which we tackle in this section:

- The type of type-safe `cast` is not general enough for some situations. We show why it should be generalised, and how, in Section 7.2.
- Type-safe `cast` works on *types* but not on *type constructors*. This limitation is important as we show in Section 7.3, where we also describe how the restriction can be lifted.

We use the term generic function "extension" for the accommodation of type-specific cases. We do not use the term "specialisation" to avoid any confusion with compile-time specialisation of generic functions in other approaches. Our approach uses fixed code and run-time type tests. As a separate matter, however, our dynamic code can, if desired, be specialised like any other type-class-overloaded function, to produce type-test-free residual code.

7.1 Monomorphic function extension

In our earlier paper [16], we described a function `extQ` that can extend (or, over-ride) a fully-generic query with a type-specific query. This allows us to refine `gshow` as follows:

```
gshow :: Data a => a -> String
gshow = gshow_help `extQ` showString

gshow_help :: Data a => a -> String
gshow_help t
   =    "("
     ++ showConstr (toConstr t)
     ++ concat (intersperse " " (gmapQ gshow t))
     ++  ")"

showString :: String -> String
showString s = "\"" ++ concat (map escape s) ++ "\""
             where
               escape '\n' = "\\n"
               ...etc...
               escape other_char = [other_char]
```

Here, the type-specific `showString` over-rides the fully-generic `gshow_help` to make the combined function `gshow`. Notice the mutual recursion between `gshow` and `gshow_help`. The function `extQ` is defined in the generics library as follows:

```
extQ :: (Typeable a, Typeable b)
     => (a -> r) -> (b -> r) -> (a -> r)
extQ fn spec_fn arg
  = case cast arg of
      Just arg' -> spec_fn arg'
      Nothing   -> fn      arg
```

The function ``(gshow_help `extQ` showString)`` behaves like the monomorphic `showString` if given a `String`, and like the polymorphic function `gshow_help` otherwise. To this end, `extQ` uses a type-safe `cast` operator, which is regarded as a primitive of the following type:

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

If the cast from `a` to `b` succeeds, one obtains a datum of the form `Just ...`, and `Nothing` otherwise. The constraints on the argument and result type of `cast` highlight that `cast` is not a parametrically polymorphic function. We rather require the types `a` and `b` to be instances of the class `Typeable`, a superclass of `Data`:[5]

```
class Typeable a where
  typeOf :: a -> TypeRep
```

Given a typeable value `v`, the expression `(typeOf v)` computes the type representation (`TypeRep`) of `v`. Like `dataTypeOf`, `typeOf` never inspects its argument. Type representations admit equality, which is required to coincide with nominal type equivalence. One specific implementation of type-safe `cast` is then to trivially guard an unsafe coercion by type equivalence. This and other approaches to casting are discussed at length in [16]. In what follows, we are merely interested in generalising the *type* of `cast`.

7.2 Generalising `cast`

The scheme that we used for extending generic *queries* is specific to queries. It cannot be reused as is for generic *transformations*:

```
extT :: (Typeable a, Typeable b)
     => (a -> a) -> (b -> b) -> (a -> a)
extT fn spec_fn arg
  = case cast arg of   -- WRONG
      Just arg' -> spec_fn arg'
      Nothing   -> fn      arg
```

The trouble is that the result of `spec_fn arg'` has a different type than the call `fn arg`. Hence, `extT` must be defined in a different style than `extQ`. One option is to cast the *function* `spec_fn` rather than the *argument* `arg`:

[5] We use two separate classes `Data` and `Typeable` to encourage well-bounded polymorphism. That is, the class `Typeable` supports nominal type representations, just enough to do cast and dynamics. The class `Data` is about structure of terms and data types.

```
extT fn spec_fn arg
  = case cast spec_fn of   -- RIGHT
      Just spec_fn' -> spec_fn' arg
      Nothing       -> fn       arg
```

This time, the `cast` compares the type of `spec_fn` with that of `fn`, and uses the former when the type matches. The only infelicity is that we thereby compare the representations of the types `a->a` and `b->b`, when all we *really* want to do is compare the representations of the types `a` and `b`. This infelicity becomes more serious when we move to *monadic* transforms:

```
extM :: ( ??? ) => (a -> m a) -> (b -> m b) -> (a -> m a)
extM fn spec_fn arg
  = case cast spec_fn of
      Just spec_fn' -> spec_fn' arg
      Nothing       -> fn       arg
```

Now, we need to construct the representation of `a -> m a`, and hence `m a` must be `Typeable` too! So the `(...???...)` must be filled in thus:

```
extM :: (Typeable a, Typeable b,
         Typeable (m a), Typeable (m b))
     => (a -> m a) -> (b -> m b) -> (a -> m a)
```

Notice the `Typeable` constraints on `(m a)` and `(m b)`, which should not be required. The type of `cast` is too specific. The primitive that we *really* want is `gcast` — generalised `cast`:

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

Here `c` is an arbitrary type constructor. By replacing `cast` by `gcast` in `extT` and `extM`, and instantiating `c` to $\Lambda a.a$`->`a, and $\Lambda a.a$ `-> m` a respectively, we can achieve the desired effect.

But wait! Haskell does not support higher-order unification, so how can we instantiate `c` to these type-level functions? We resort to the standard technique, which uses a `newtype` to explain to the type engine which instantiation is required. Here is `extM`:

```
extM :: (Typeable a, Typeable b)
     => (a -> m a) -> (b -> m b) -> (a -> m a)
extM fn spec_fn arg
  = case gcast (M spec_fn) of
      Just (M spec_fn') -> spec_fn' arg
      Nothing           -> fn       arg

newtype M m a = M (a -> m a)
```

Here, `(M spec_fn)` has type `(M m a)`, and that fits the type of `gcast` by instantiating `c` to `M m`. We can rewrite `extQ` and `extT` to use `gcast`, in exactly the same way:

```
extQ fn spec_fn arg
  = case gcast (Q spec_fn) of
      Just (Q spec_fn') -> spec_fn' arg
      Nothing           -> fn       arg

newtype Q r a = Q (a -> r)

extT fn spec_fn arg
  = case gcast (T spec_fn) of
      Just (T spec_fn') -> spec_fn' arg
      Nothing           -> fn       arg

newtype T a = T (a -> a)
```

As with `cast` before, `gcast` is best regarded as a built-in primitive, but in fact `gcast` replaces `cast`. Our implementation of `cast`, discussed at length in [16], can be adopted directly for `gcast`. The only difference is that `gcast` neglects the type constructor `c` in the test for type equivalence [17].

This generalisation, from `cast` to `gcast`, is not a new idea. Weirich [22] uses the same generalisation, from `cast` to `cast'` in her case, albeit using structural rather than nominal type equality. We used a very similar pattern in our earlier paper, when we generalised `gmapQ`, `gmapT` and `gmapM` to produce the function `gfoldl` [16].

7.3 Polymorphic function extension

The function `extQ` allows us to extend a generic function at a particular *monomorphic* type, but not at a *polymorphic* type. For example, as it stands `gshow` will print lists in prefix form `"(: 1 (: 2 : []))"`. How could we print lists in distfix notation, thus `"[1,2]"`?

Our raw material must be a *list-specific*, but still *element-generic* function that prints lists in distfix notation:

```
gshowList :: Data b => [b] -> String
gshowList xs
  = "[" ++ concat (intersperse "," (map gshow xs)) ++ "]"
```

Now we need to extend `gshow_help` with `gshowList` — but `extQ` has the wrong type. Instead, we need a higher-kinded version of `extQ`, which we call `ext1Q`:

```
ext1Q :: (Typeable a, Typeable1 t)
      => (a -> r)
      -> (forall b. Data b => t b -> r)
      -> (a -> r)

gshow :: Data a => a -> String
gshow = gshow_help `ext1Q` gshowList
                   `extQ`  showString
```

Here, `ext1Q` is quantified over a type *constructor* `t` of kind `*->*`, and hence we need a new type class `Typeable1`: Haskell sadly lacks kind polymorphism! (This would require a non-trivial language extension.) We discuss `Typeable1` in Section 7.4.

To define `ext1Q` we can follow exactly the same pattern as for `extQ`, above, but using a different `cast` operator:

```
ext1Q fn spec_fn arg
  = case dataCast1 (Q spec_fn) of
      Just (Q spec_fn') -> spec_fn' arg
      Nothing           -> fn       arg
newtype Q r a = Q (a -> r)
```

Here, we need (another) new `cast` operator, `dataCast1`. Its type is practically forced by the definition of `ext1Q`:

```
dataCast1 :: (Typeable1 s, Data a)
          => (forall b. Data b => c (s b))
          -> Maybe (c a)
```

It is absolutely necessary to have the `Data` constraint in the argument to `dataCast1`. For example, this will not work at all:

```
bogusDataCast1 :: (Typeable1 s, Typeable a)
               => (forall b. c (s b))
               -> Maybe (c a)
```

It will not work because the argument is required to be completely polymorphic in `b`, and our desired arguments, such as `showList` are not; they need the `Data` constraint. That is why the "`Data`" appears in the name `dataCast1`.

How, then are we to implement `dataCast1`? We split the implementation into two parts. The first part performs the type test (Section 7.4), while the second instantiates the argument to `dataCast1` (Section 7.5).

7.4 Generalising `cast` again

First, the type test. We need a primitive `cast` operator, `gcast1`, that matches the *type constructor* of the argument, rather than the *type*. Here is its type along with that of `gcast` for comparison:

```
gcast1 :: (Typeable1 s, Typeable1 t)  -- New
       => c (s a) -> Maybe (c (t a))

gcast :: (Typeable a, Typeable b)     -- For comparison
      => c a -> Maybe (c b)
```

The role of `c` is unchanged. The difference is that `gcast1` compares the type constructors `s` and `t`, instead of the types `a` and

b. As with our previous generalisation, from `cast` to `gcast`, the `Typeable` constraints concern only the differences between the two types whose common shape is `(c (• a))`. The implementation of `gcast1` follows the same trivial scheme as before [16, 17].

The new class `Typeable1` is parameterised over type constructors, and allows us to extract a representation of the type constructor:

```
class Typeable1 s where
  typeOf1 :: s a -> TypeRep

instance Typeable1 [] where
  typeOf1 _ = mkTyConApp (mkTyCon "Prelude.List") []

instance Typeable1 Maybe where
  typeOf1 _ = mkTyConApp (mkTyCon "Prelude.Maybe") []
```

The operation `mkTyCon` constructs type-constructor representations. The operation `mkTyConApp` turns the latter into potentially incomplete type representations subject to further type applications. There is a single `Typeable` instance for all types with an outermost type constructors of kind `*->*`:

```
instance (Typeable1 s, Typeable a)
      => Typeable (s a) where
  typeOf x = typeOf1 x `mkAppTy` typeOf (undefined :: a)
```

(Notice the use of a scoped type variable here. Also, generic instances are not Haskell 98 compliant. One could instead use one instance per type constructor of kind `*->*`.) The function `mkAppTy` applies a type-constructor representation to an argument-type representation. In the absence of kind polymorphism, we sadly need a distinct `Typeable` class for each kind of type constructor. For example, for binary type constructors we have:

```
class Typeable2 s where
  typeOf2 :: s a b -> TypeRep

instance (Typeable2 s, Typeable a)
      => Typeable1 (s a) where
  typeOf1 x = typeOf2 x `mkAppTy` typeOf (undefined :: a)
```

One might worry about the proliferation of `Typeable` classes, but in practice this is not a problem. First, we are primarily interested in type constructors whose arguments are themselves of kind `*`, because the `Data` class only makes sense for *types*. Second, the arity of type constructors is seldom large.

7.5 Implementing `dataCast1`

Our goal is to implement `dataCast1` using `gcast1`:

```
dataCast1 :: (Typeable1 s, Data a)
          => (forall b. Data b => c (s b))
          -> Maybe (c a)

gcast1 :: (Typeable1 s, Typeable1 t)
       => c (s a) -> Maybe (c (t a))
```

There appear to be two difficulties. First, `dataCast1` must work over *any* type `(c a)`, whereas `gcast1` is restricted to types of form `(c (t a))`. Second, `dataCast1` is given a polymorphic argument which it must instantiate by applying it to a dictionary for `Data a`. Both these difficulties can, indeed must, be met by making `dataCast1` into a member of the `Data` class itself:

```
class Typeable a => Data a where
  ...
  dataCast1 :: Typeable1 s
            => (forall a. Data a => c (s b))
            -> Maybe (c a)
```

Now in each instance declaration we have available precisely the necessary `Data` dictionary to instantiate the argument. All `dataCast1` has to do is to instantiate `f`, and pass the instantiated version on to `gcast1` to perform the type test, yielding the following, mysteriously simple implementation:

```
instance Data a => Data [a] where
  ...
  dataCast1 f = gcast1 f
```

The instances of `dataCast1` for type constructors of kind other than `*->*` returns `Nothing`, because the type is not of the required form.

```
instance Data Int where
  ...
  dataCast1 f = Nothing
```

Just as we need a family of `Typeable` classes, so we need a family of `dataCast` operators with an annoying but unavoidable limit.

7.6 Generic function extension — summary

Although this section has been long and rather abstract, the concrete results are simple to use. We have been able to generalise `extQ`, `extT`, `extM` (and any other variants you care to think of) so that they handle *polymorphic* as well as monomorphic cases. The new operators are easy to use — see the definition of `gshow` in Section 7.3 — and are built on an interesting and independently-useful generalisation of the `Typeable` class. All the instances for `Data` and `Typeable` are generated automatically by the compiler, and need never be seen by the user.

8 Related work

The position of the *Scrap your boilerplate* approach within the generic programming field was described in the original paper. Hence, we will focus on related work regarding the new contributions of the present paper: type reflection (Section 5), zipping combinators (Section 6), and generic function extension (Section 7).

Our type reflection is a form of introspection, i.e., the structure of types can be observed, including names of constructors, fields, and types. In addition, terms can be constructed. This is similar to the reflection API of a language like Java, where attributes and method signatures can be observed, and objects can be constructed from class names. The sum-of-products approach to generic programming abstracts from everything except type structure. In the pure sum-of-products setup, one cannot define generic read and show functions. There are non-trivial refinements, which enrich induction on type structure with cases for constructor applications and labelled components [7, 4, 8]. In our approach, reflective information travels silently with the `Data` dictionaries that go with any data value. This is consistent with the aspiration of our approach to define generic functions without reference to a universal representation, and without compile-time specialisation. Altenkirch and McBride's generic programming with dependent types [1] suggests that reflective data can also be represented as types, which is more typeful than our approach.

Zipping is a well-known generic operation [12, 4, 13]. Our development shows that zippers can be defined generically as curried folds, while taking advantage of higher-order generic functions. Defining zippers by pattern matching on two parameters instead, would require a non-trivial language extension. In the sum-of-product approach, zippers perform polymorphic pattern matching on the two incoming data structures simultaneously. To this end, the generic function is driven by the type structure of a *shared* type constructor, which implies dependently polymorphic argument types [12, 4]. Altenkirch and McBride's generic programming with dependent types [1] indicates that argument type dependencies as in zipping can be captured accordingly with dependent types if this is intended. Their approach also employs a highly parametric fold operator that is readily general for multi-parameter traversal. The pattern calculus (formerly called constructor calculus) by Barry Jay [13], defines zipping-like operations by simultaneous pattern matching on two arbitrary *constructor applications*. Like in our zippers, the argument types are independently polymorphic.

Customisation of generic functions for specific types is an obvious desideratum. In Generic Haskell, generic function definitions can involve some sort of ad-hoc or default cases [7, 5, 4, 19]. Our approach narrows down generic function extension to the very simple construct of a nominal type cast [16]. However, our original paper facilitated generic function extension with only monomorphic cases as a heritage of our focus on term traversal. The new development of Section 7 generalised from monomorphic to polymorphic cases in generic function extension. This generality of generic function extension is also accommodated by Generic Haskell, but rather at a static level relying on a dedicated top-level declaration form for generic functions. By contrast, our generic function extension facilitates *higher-order* generic functions.

In a very recent paper [8], Hinze captures essential idioms of Generic Haskell in a Haskell 98-based model, which requires absolutely no extensions. Nevertheless, the approach is quite general. For instance, it allows one to define generic functions that are indexed by type constructors. This work shares our aspiration of lightweightness as opposed to the substantial language extension of Generic Haskell [7, 5, 4, 19]. Hinze's lightweight approach does not support some aspects of our system. Notably, Hinze's generic functions are not higher-order; and generic functions operate on a representation type. Furthermore, the approach exhibits a limitation related to generic function extension: the *class* for generics would need to be adapted for each new type or type constructor that requires a specific case.

9 Conclusion

We have completed the *Scrap your boilerplate* approach to generic programming in Haskell, which combines the following attributes:

Lightweight: the approach requires two independently-useful language extensions to Haskell 98 (higher-rank types and type-safe cast), after which everything can be implemented as a library. A third extension, extending the `deriving` clause to handle `Data` and `Typeable` is more specific to our approach, but this code-generation feature is very non-invasive.

General: the approach handles regular data types, nested data types, mutually-recursive data types, type constructor parameterised in additional types; and it handles single and multi-parameter term traversal, as well as term building.

Versatile: the approach supports higher-order generic programming, reusable definitions of traversal strategies, and overriding of generic functions at specified types. There is no closed world assumption regarding user-defined data types.

Direct: generic functions are directly defined on Haskell data types without detouring to a uniform representation type such as sums-of-products. Also, Haskell's nominal type equivalence is faithfully supported, as opposed to more structurally-defined generic functions.

Well integrated and supported: everything we describe is implemented in GHC and supported by a Haskell generics library.

Acknowledgements. We gratefully acknowledge very helpful comments and suggestions by four anonymous ICFP 2004 referees as well as by Olaf Chitil, Andres Löh, and Simon Marlow.

10 References

[1] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Generic Programming*, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, July 2002.

[2] F. Atanassow, D. Clarke, and J. Jeuring. UUXML: A Type-Preserving XML Schema-Haskell Data Binding. In B. Jayaraman, editor, *Practical Aspects of Declarative Languages: 6th International Symposium, PADL 2004, Dallas, TX, USA, June 18-19, 2004. Proceedings*, volume 3057 of *LNCS*, pages 71–85. Springer-Verlag, May 2004.

[3] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In ICFP00 [11], pages 268–279.

[4] D. Clarke, J. Jeuring, and A. Löh. The Generic Haskell User's Guide, 2002. Version 1.23 — Beryl release.

[5] D. Clarke and A. Löh. Generic Haskell, Specifically. In J. Gibbons and J. Jeuring, editors, *Proc. of the IFIP TC2 Working Conference on Generic Programming*. Kluwer Academic Publishers, 2003.

[6] J. Harm and R. Lämmel. Two-dimensional Approximation Coverage. *Informatica*, 24(3):355–369, 2000.

[7] R. Hinze. A generic programming extension for Haskell. In *Proc. 3rd Haskell Workshop, Paris, France*, 1999. Technical report of Universiteit Utrecht, UU-CS-1999-28.

[8] R. Hinze. Generics for the masses. In these proceedings, 2004.

[9] R. Hughes. A novel representation of lists and its application to the function reverse. *Information Processing Letters*, 22, 1986.

[10] G. Hutton and E. Meijer. Functional pearl: Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[11] *ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, Montreal, Sept. 2000. ACM.

[12] P. Jansson and J. Jeuring. PolyLib—A library of polytypic functions. In R. Backhouse and T. Sheard, editors, *Proc. of Workshop on Generic Programming, WGP'98, Marstrand, Sweden*. Dept. of Comp. Science, Chalmers Univ. of Techn. and Göteborg Univ., June 1998.

[13] C. B. Jay. The pattern calculus. `http://www-staff.it.uts.edu.au/~cbj/Publications/pattern_calculus.ps`, 2003. (accepted for publication by ACM TOPLAS.).

[14] P. W. M. Koopman, A. Alimarine, J. Tretmans, and M. J. Plasmeijer. Gast: Generic Automated Software Testing. In R. Pena and T. Arts, editors, *Implementation of Functional Languages, 14th International Workshop, IFL 2002, Madrid, Spain, September 16-18, 2002, Revised Selected Papers*, volume 2670 of *LNCS*, pages 84–100. Springer-Verlag, 2003.

[15] R. Lämmel and J. Harm. Test case characterisation by regular path expressions. In E. Brinksma and J. Tretmans, editors, *Proc. Formal Approaches to Testing of Software (FATES'01)*, Notes Series NS-01-4, pages 109–124. BRICS, Aug. 2001.

[16] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, Mar. 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[17] The "*Scrap your boilerplate*" web site: examples, browsable library, papers, background, 2003–2004. `http://www.cs.vu.nl/boilerplate/`.

[18] D. Lelewer and D. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, Sept. 1987.

[19] A. Löh, D. Clarke, and J. Jeuring. Dependency-style Generic Haskell. In C. Norris and J. J. B. Fenwick, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP-03)*, volume 38, 9 of *ACM SIGPLAN Notices*, pages 141–152, New York, Aug. 25–29 2003. ACM Press.

[20] S. Peyton Jones and M. Shields. Practical type inference for higher-rank types. Unpublished manuscript, 2004.

[21] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 148–159, Paris, Sept. 1999. ACM.

[22] S. Weirich. Type-safe cast. In ICFP00 [11], pages 58–67.

[23] I. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *CACM*, 30(6):520–540, June 1987.