Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming

Ralf Lämmel
Vrije Universiteit, Amsterdam

Simon Peyton Jones
Microsoft Research, Cambridge

Abstract

We describe a design pattern for writing programs that traverse data structures built from rich mutually-recursive data types. Such programs often have a great deal of "boilerplate" code that simply walks the structure, hiding a small amount of "real" code that constitutes the reason for the traversal.

Our technique allows most of this boilerplate to be written once and for all, or even generated mechanically, leaving the programmer free to concentrate on the important part of the algorithm. These generic programs are much more adaptive when faced with data structure evolution because they contain many fewer lines of type-specific code.

Our approach is simple to understand, reasonably efficient, and it handles all the data types found in conventional functional programming languages. It makes essential use of rank-2 polymorphism, an extension found in some implementations of Haskell. Further it relies on a simple type-safe cast operator.

Categories and Subject Descriptors

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

General Terms

Design, Languages

Keywords

Generic programming, traversal, rank-2 types, type cast

1 Introduction

Suppose you have to write a function that traverses a rich, recursive data structure representing a company's organisational structure, and increases the salary of every person in the structure by 10%. The interesting bit of this algorithm is performing the salary-increase — but the code for the function is probably dominated by "boilerplate" code that recurses over the data structure to find the specified department as spelled out in Section 2. This is not an unusual situation. On the contrary, performing queries or transformations over rich data structures, nowadays often arising from XML schemata, is becoming increasingly important.

Boilerplate code is tiresome to write, and easy to get wrong. Moreover, it is vulnerable to change. If the schema describing the company's organisation changes, then so does every algorithm that recurses over that structure. In small programs which walk over one or two data types, each with half a dozen constructors, this is not much of a problem. In large programs, with dozens of mutually recursive data types, some with dozens of constructors, the maintenance burden can become heavy.

Generic programming techniques aim to eliminate boilerplate code. There is a large literature, as we discuss in Section 9, but much of it is rather theoretical, requires significant language extensions, or addresses only "purely-generic" algorithms. In this paper, we present a simple but powerful design pattern for writing generic algorithms in the strongly-typed lazy functional language Haskell. Our technique has the following properties:

- It makes the application program adaptive in the face of data type (or schema) evolution. As the data types change, only two functions have to be modified, and those functions can easily be generated because they are not application-specific.
- It is simple *and* general. It copes with arbitrary data-type structure without fuss, including parameterised, mutually-recursive, and nested types. It also subsumes other styles of generic programming such as term rewriting strategies.
- It requires two extensions to the Haskell type system, namely (a) rank-2 types and (b) a form of type-coercion operator. However these extensions are relatively modest, and are independently useful; they have both been available in two popular implementations of Haskell, GHC and Hugs, for some time.

Our contribution is one of synthesis: we put together some relatively well-understood ideas (type-safe cast, one-layer maps) in an innovative way, to solve a practical problem of increasing importance. The paper should be of direct interest to programmers, and library designers, but also to language designers because of the further evidence for the usefulness of rank-2 polymorphic types.

The code for all the examples is available online at:

`http://www.cs.vu.nl/Strafunski/gmap/`

The distribution comes with generative tool support to generate all datatype-specific boilerplate code. Our benchmarks show that it is possible to get the run-time performance of typical generic programs reasonably close to the hand-coded boilerplate-intensive counterparts (Section 10).

TLDI'03, January 18, 2003, New Orleans, Louisiana, USA.

2 The problem

We begin by characterising the problem we are addressing. Consider the following data types that describe the organisational structure of a company. A company is divided into departments which in turn have a manager, and consists of a collection of sub-units. A unit is either a single employee or a department. Both managers and ordinary employees are persons receiving a salary. That is:

```
data Company  = C [Dept]
data Dept     = D Name Manager [SubUnit]
data SubUnit  = PU Employee | DU Dept
data Employee = E Person Salary
data Person   = P Name Address
data Salary   = S Float
type Manager  = Employee
type Name     = String
type Address  = String
```

Here is a small company represented by such a data structure:

```
genCom :: Company
genCom = C [D "Research" ralf [PU joost, PU marlow],
            D "Strategy" blair []]

ralf, joost, marlow, blair :: Employee
ralf   = E (P "Ralf"   "Amsterdam") (S 8000)
joost  = E (P "Joost"  "Amsterdam") (S 1000)
marlow = E (P "Marlow" "Cambridge") (S 2000)
blair  = E (P "Blair"  "London")    (S 100000)
```

The advent of XML has made schemata like this much more widespread, and many tools exist for translating XML schemata into data type definitions in various languages; in the case of Haskell, HaXML includes such a tool [35]. There are often many data types involved, sometimes with many constructors, and their structure tends to change over time.

Now suppose we want to increase the salary of everyone in the company by a specified percentage. That is, we must write the function:

```
increase :: Float -> Company -> Company
```

So that `(increase 0.1 genCom)` will be just like `genCom` except that everyone's salary is increased by 10%. It is perfectly straightforward to write this function in Haskell:

```
increase k (C ds) = C (map (incD k) ds)

incD :: Float -> Dept -> Dept
incD k (D nm mgr us) =
 D nm (incE k mgr) (map (incU k) us)

incU :: Float -> SubUnit -> SubUnit
incU k (PU e) = PU (incE k e)
incU k (DU d) = DU (incD k d)

incE :: Float -> Employee -> Employee
incE k (E p s) = E p (incS k s)

incS :: Float -> Salary -> Salary
incS k (S s) = S (s * (1+k))
```

Looking at this code, it should be apparent what we mean by "boilerplate". Almost all the code consists of a routine traversal of the tree. The only interesting bit is `incS` which actually increases a `Salary`. As the size of the data type increases, the ratio of interesting code to boilerplate decreases. Worse, this sort of boilerplate needs to be produced for each new piece of traversal functionality. For example, a function that finds the salary of a named individual would require a new swathe of boilerplate.

3 Our solution

Our goal, then, is to write `increase` without the accompanying boilerplate code. To give an idea of what is to come, here is the code for `increase`:

```
increase :: Float -> Company -> Company
increase k = everywhere (mkT (incS k))
```

And that is it! This code is formed from four distinct ingredients:

- The function `incS` (given in Section 2) is the "interesting part" of the algorithm. It performs the arithmetic to increase a `Salary`.
- The function `mkT` makes a *type extension* of `incS` (read `mkT` as "make a transformation"), so that it can be applied to any node in the tree, not just `Salary` nodes. The type-extended function, `mkT (incS k)`, behaves like `incS` when applied to a `Salary` and like the identity function when applied to any other type. We discuss type extension in Section 3.1.
- The function `everywhere` is a *generic traversal combinator* that applies its argument function to every node in the tree. In this case, the function is the type-extended `incS` function, which will increase the value of `Salary` nodes and leave all others unchanged. We discuss generic traversal in Sections 3.2 and 3.3.
- Both `mkT` and `everywhere` are overloaded functions, in the Haskell sense, over the classes `Typeable` and `Term` (to be introduced shortly). For each data type involved (`Company`, `Dept`, `Person`, etc.) the programmer must therefore give an `instance` declaration for the two classes. However these instances are, as we shall see in Sections 3.2 and 8, extremely simple — in fact, they are "pure boilerplate" — and they can easily be generated mechanically. The software distribution that comes with the paper includes a tool to do just that.

The following sections fill in the details of this sketch.

3.1 Type extension

The first step is to extend a function, such as `incS`, that works over a single type t, to a function that works over many types, but is the identity at all types but t. The fundamental building-brick is a type-safe `cast` operator the type of which involves a Haskell class `Typeable` of types that can be subject to a cast:

```
-- An abstract class
class Typeable

-- A type-safe cast operator
cast :: (Typeable a, Typeable b) => a -> Maybe b
```

This `cast` function takes an argument `x` of type `a`. It makes a run-time test that compares the types `a` and `b`; if they are the same type, `cast` returns `Just x`; if not, it returns `Nothing`.[1] For example, here is an interactive `GHCi` session:

```
Prelude> (cast 'a') :: Maybe Char
Just 'a'
Prelude> (cast 'a') :: Maybe Bool
Nothing
Prelude> (cast True) :: Maybe Bool
Just True
```

The type signature in the above samples gives `cast` its result context, `Typeable b`, so it knows what the result type must be; without that, it cannot do the type test. Because the type class `Typeable` constrains the types involved, `cast` is not completely polymorphic: both argument and result types must be instances of the class `Typeable`.

Type-safe cast can be integrated with functional programming in various ways, preferably by a language extension. In fact, it is

[1] In many languages a "cast" operator performs a *representation change* as well as *type change*. Here, `cast` is operationally the identity function; it only makes a type change.

well-known folk lore in the Haskell community that much of the functionality of `cast` can be programmed in standard Haskell. In Section 8, we provide a corresponding Haskell-encoding that can be regarded as a reference implementation for type-safe cast. This will clarify that a corresponding extension turns out to be a modest one. For the coming sections we will simply assume that `cast` is available, and that every type is an instance of `Typeable`.

Given `cast`, we can write `mkT`, which we met in Section 3:

```
mkT :: (Typeable a, Typeable b)
    => (b -> b) -> a -> a
mkT f = case cast f of
          Just g  -> g
          Nothing -> id
```

That is, `mkT f x` applies `f` to `x` if `x`'s type is the same as `f`'s argument type, and otherwise applies the identity function to `x`. Here are some examples:

```
Prelude> (mkT not) True
False
Prelude> (mkT not) 'a'
'a'
```

"`mkT`" is short for "make a transformation", because it constructs a generic transformation function. We can use `mkT` to lift `incS`, thus:

```
inc :: Typeable a => Float -> a -> a
inc k = mkT (incS k)
```

So `inc` is applicable to any type that is an instance of `Typeable` but we ultimately aim at a function that applies `inc` to *all nodes* in a tree. This necessitates generic traversal.

3.2 One-layer traversal

Our approach to traversal has two steps: for each data type we write a single function, `gmapT`, that traverses values of that type; then we build a variety of recursive traversals from `gmapT`. In the context of Haskell, we overload `gmapT` using a type class, `Term`:

```
class Typeable a => Term a where
  gmapT :: (forall b. Term b => b -> b) -> a -> a
```

The intended behaviour is this: `gmapT` takes a generic transformation (such as `inc k`) and applies it to all the *immediate* children of the value. It is easiest to understand this idea by example. Here is the `instance` declaration for `Employee`:

```
instance Term Employee where
  gmapT f (E per sal) = E (f per) (f sal)
```

Here we see clearly that `gmapT` simply applies `f` to the immediate children of `E`, namely `per` and `sal`, and rebuilds a new `E` node.

There are two things worth mentioning regarding the type of `gmapT` and its hosting class `Term`. Firstly, `gmapT` has a non-standard type: its first argument is a *polymorphic* function, of type `forall b. Term b => b -> b`. Why? Because it is applied to both `per` and `sal` in the `instance` declaration, and those two fields have different types. Haskell 98 would reject the type of `gmapT`, but rank-2 types like these have become quite well-established in the Haskell community. We elaborate in Section 9.1. Secondly, note the recursion in the class declaration of `Term`. The member signature for `gmapT` refers to `Term` via a class constraint.

Obviously, we can provide a simple schematic definition for `gmapT` for arbitrary terms `C t1 ... tn`:

```
gmapT f (C t1 ... tn) = C (f t1) ... (f tn)
```

When the node has no children, `gmapT` has no effect. Hence the `Term` instance for `Bool` looks like this:

```
instance Term Bool where
  gmapT f x = x
```

The important thing to notice is that `gmapT` only applies `f` to the *immediate* children of the node as opposed to any kind of recursive traversal. Here, for example, is the `Term` instance for lists, which follows exactly the same pattern as the instance for `Employee`:

```
instance Term a => Term [a] where
  gmapT f []     = []
  gmapT f (x:xs) = f x : f xs
```

Notice the "`f xs`" for the tail — not "`gmapT f xs`"; `gmapT` traverses one layer only, unlike the common recursive `map` function.

3.3 Recursive traversal

Even though `gmapT` has this one-layer-only behaviour, we can synthesise a variety of recursive traversals from it. Indeed, as we shall see, it is precisely its one-layer behaviour that makes this variety easy to capture.

For example, the `everywhere` combinator applies a transformation to every node in a tree:

```
-- Apply a transformation everywhere, bottom-up
everywhere :: Term a
           => (forall b. Term b => b -> b)
           -> a -> a
everywhere f x = f (gmapT (everywhere f) x)
```

We can read this function as follows: first apply `everywhere f` to all the children of `x`, and then apply `f` to the result. The recursion is in the definition of `everywhere`, not in the definition of `gmapT`.[2]

The beautiful thing about building a recursive traversal strategy out of non-recursive `gmapT` is that we can build many different strategies using a single definition of `gmapT`. As we have seen, `everywhere` works bottom-up, because `f` is applied after `gmapT` has processed the children. It is equally easy to do top-down:

```
-- Apply a transformation everywhere, top-down
everywhere' :: Term a
            => (forall b. Term b => b -> b)
            -> a -> a
everywhere' f x = gmapT (everywhere' f) (f x)
```

In the rest of this paper we will see many different recursive strategies, each of which takes a line or two to define.

This extremely elegant way of building a *recursive* traversal in two steps — first define a *one-layer* map, and then tie the recursive knot separately — is well-known folk lore in the functional programming community, e.g., when dealing with ana- and catamorphisms for regular data types such as lists [22]. For lack of better-established terminology we call it "*the non-recursive map trick*", and review it in Section 9.2.

3.4 Another example

Lest we get fixated on `increase` here is another example that uses the same design pattern. Let us write a function that flattens out a named department *d*; that is, it takes all *d*'s sub-units and makes them part of *d*'s parent department:

```
flatten :: Name -> Company -> Company
flatten d = everywhere (mkT (flatD d))

flatD :: Name -> Dept -> Dept
flatD d (D n m us)
  = D n m (concatMap unwrap us)
 where
   unwrap :: SubUnit -> [SubUnit]
   unwrap (DU (D d' m us)) | d==d' = PU m : us
   unwrap u                        = [u]
```

[2] In "point-free" notation:

```
everywhere f = f . gmapT (everywhere f)
```

The function `flatD` does the interesting work on a department: it looks at each of its sub-units, `u`, applies `unwrap` to get a list of units (usually the singleton list `[u]`), and concatenates the results.[3] When `unwrap` sees the target department (`d == d'`) it returns all its sub-units. The manager `m` is not fired, but is turned into a plain working unit, `PU m` (presumably subject to drastic subsequent salary decrease).

Again, this is *all* the code for the task. The one-line function `flatten` uses exactly the same combinators `everywhere` and `mkT` as before to "lift" `flatD` into a function that is applied everywhere in the tree.

Furthermore, if the data types change – for example, a new form of `SubUnit` is added – then the per-data-type boilerplate code must be re-generated, but the code for `increase` and `flatten` is unchanged. Of course, if the number of fields in a `Dept` or `SubUnit` changed, then `flatD` would have to change too, because `flatD` mentions the `DU` and `D` constructors explicitly. But that is not unreasonable; if a `Dept`'s units were split into two lists, say, one for people and one for sub-departments, the algorithm really would have to change.

3.5 Summary

We have now completed an initial description of our new design pattern. To summarise, an application is built from three chunks of code:

Programmer-written: a short piece of code for the particular application. This typically consists of (a) a code snippet to do the real work (e.g., `incS`) and (b) the application of some strategy combinators that lift that function to the full data type, and specify the traversal scheme.

Mechanically-generated: for each data type, two `instance` declarations, one for class `Typeable` and one for class `Term`. The former requires a fixed amount of code per data type (see Section 8). The latter requires one line of code per constructor, as we have seen. Because the two kinds of `instance` declarations take a very simple, regular form, they can readily be generated mechanically.

Library: a fixed library of combinators, such as `mkT` and `everywhere`. The programmer can readily extend this library with new forms of traversal.

One way to generate the `instance` declarations is to use the DrIFT pre-processor [38]. Furthermore, derivable type classes [11] (almost) can do the job, or Template Haskell [30]. The software distribution that comes with the paper includes a customised version of DrIFT. However, mechanical support is not absolutely necessary: writing this boilerplate code by hand is not onerous and it still pays off because it is a one-off task.

The rest of the paper consists of an elaboration and generalisation of the ideas we have presented. The examples we have seen so far are all generic *transformations* that take a `Company` and produce a new `Company`. It turns out that two other forms of generic algorithms are important: generic *queries* (Section 4) and *monadic transformations* (Section 5). After introducing these forms, we pause to reflect and generalise on the ideas (Section 6), before showing that the three forms of algorithm can all be regarded as a form of fold operation (Section 7). Lastly, we return to the type-safe cast operator in Section 8.

[3] `concatMap :: (a->[b]) -> [a] -> [b]` maps a function over a list and concatenates the results.

4 Queries

Thus far we have concentrated on generic *transformations*. We recall the corresponding type scheme:

```
forall a. Term a => a -> a
```

There is a second interesting class of generic programs that we call *generic queries*. A generic query has a type of the following form:

```
forall a. Term a => a -> R
```

Here `R` is some fixed result type. For example, suppose we wanted to compute the salary bill of the company; we would need a function of the following type:

```
salaryBill :: Company -> Float
```

Here `Float` is the fixed result type `R`.

4.1 Implementing queries

Our general approach is exactly the same as before: we use type extension to lift the interesting part of the function into a polymorphic function; for each data types we give a single overloaded traversal function; and we build `salaryBill` from these two pieces. Here is the code, which looks very similar to that for `increase`:

```
salaryBill :: Company -> Float
salaryBill = everything (+) (0 ‘mkQ‘ billS)

billS :: Salary -> Float
billS (S f) = f
```

The interesting part of `salaryBill` is the function `billS` that applies to a `Salary`. To lift `billS` to arbitrary types, we use `mkQ`, a cousin of `mkT`:

```
mkQ :: (Typeable a, Typeable b)
    => r -> (b -> r) -> a -> r
(r ‘mkQ‘ q) a = case cast a of
                  Just b  -> q b
                  Nothing -> r
```

That is, the query `(r ‘mkQ‘ q)` behaves as follows when applied to an argument `a`: if `a`'s type is the same as `q`'s argument type, use `q` to interrogate a; otherwise return the default value `r`. To illustrate, here are some examples of using `mkQ` in an interactive session (recall that `ord` has type `Char -> Int`):

```
Prelude> (22 ‘mkQ‘ ord) ’a’
97
Prelude> (22 ‘mkQ‘ ord) ’b’
98
Prelude> (22 ‘mkQ‘ ord) True
22
```

The next step is to extend the `Term` class with a function `gmapQ` that applies the specified query function and makes a list of the results:

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

The instances of `gmapQ` are as simple as those for `gmapT`:

```
instance Term Employee where
  gmapT = ... as before ...
  gmapQ f (E p s) = [f p, f s]

instance Term a => Term [a] where
  gmapT = ... as before ...
  gmapQ f []     = []
  gmapQ f (x:xs) = [f x, f xs]

instance Term Bool where
  gmapT x = ... as before ...
  gmapQ x = []
```

Just as with `gmapT`, notice that there is no recursion involved (it is a one-layer operator), and that the function has a rank-2 type.

Now we can use `gmapQ` to build the `everything` combinator that performs the recursive traversal. Like any fold, it needs an operator `k` to combine results from different sub-trees:

```
-- Summarise all nodes in top-down, left-to-right
everything :: Term a
           => (r -> r -> r)
           -> (forall a. Term a => a -> r)
           -> a -> r
everything k f x
  = foldl k (f x) (gmapQ (everything k f) x)
```

Here we see that `everything` processes the children of `x`, giving a list of results; and then combines those results using the ordinary list function `foldl`, with the operator `k` as the combiner. The `(f x)` is the result of applying the query to `x` itself, and that result is included in the `foldl`. And that concludes the definition of `salaryBill`.

4.2 Other queries

By changing the query function and combining operator we can easily query for a single value rather than combining values from all nodes in the tree. For example, here is how to extract a named department from the company data structure:

```
find :: Name -> Company -> Maybe Dept
find n = everything orElse (Nothing ‘mkQ‘ findD n)

findD :: String -> Dept -> Maybe Dept
findD n d@(D n′ _ _)
      | n == n′   = Just d
      | otherwise = Nothing

orElse :: Maybe a -> Maybe a -> Maybe a
x ‘orElse‘ y = case x of
                 Just _  -> x
                 Nothing -> y
```

The use of `foldl` in `everything` means that `find` will find the leftmost, shallowest department with the specified name. It is easy to make variants of `everything` that would find the right-most, deepest, or whatever. Laziness plays a role here: once a department of the specified name has been found, traversal will cease.

5 Monadic transformation

As well as transformations (Section 3) and queries (Section 4) there is a third useful form of generic algorithm, namely a *monadic transformation*. For example, suppose we wanted to process a `Company` structure discarding the old `Salary` values, and filling in new ones by looking up the employee’s name in an external database. That means there is input/output involved, so the function must have type

```
lookupSalaries :: Company -> IO Company
```

This type does not fit the scheme for generic transformations or queries, so we have to re-run the same development one more time. First, we need a function `mkM` to construct basic monadic transformations:

```
mkM :: (Typeable a, Typeable b,
        Typeable (m a), Typeable (m b),
        Monad m)
    => (b -> m b) -> a -> m a
mkM f = case cast f of
          Just g  -> g
          Nothing -> return
```

The type of `mkM` looks somewhat scary, but it simply explains all the type-representation constraints that are needed for type-safe cast. Then we need to extend once more the class `Term` to support monadic traversal:

```
class Typeable a => Term a where
  gmapT :: ... as before ...
  gmapQ :: ... as before ...

  gmapM :: Monad m
        => (forall b. Term b => b -> m b)
        -> a -> m a
```

The instances for `gmapM` are just as simple as before; they use Haskell’s `do` notation for monadic composition:

```
instance Term Employee where
  ...
  gmapM f (E p s) = do p′ <- f p
                       s′ <- f s
                       return (E p′ s′)

instance Term a => Term [a] where
  ...
  gmapM f []     = return []
  gmapM f (x:xs) = do x′  <- f x
                      xs′ <- f xs
                      return (x′:xs′)
```

Now we can make an `everywhereM` combinator:

```
everywhereM :: (Monad m, Term a)
            => (forall b. Term b => b -> m b)
            -> a -> m a
everywhereM f x = do x′ <- gmapM (everywhereM f) x
                     f x′
```

Finally, we can write `lookupSalaries` as follows:

```
lookupSalaries = everywhereM (mkM lookupE)

lookupE :: Employee -> IO Employee
lookupE (E p@(P n _) _)
  = do { s <- dbLookup n; return (E p s) }

dbLookup :: Name -> IO Salary
-- Lookup the person in the external database
```

The obvious question is this: will each new application require a new variant of `gmap`? We discuss that in Section 7. Meanwhile, we content ourselves with two observations. First, `gmapT` is just a special case of `gmapM`, using the identity monad. (In fact, `gmapQ` can also be encoded using `gmapM`, although not so directly.) Second, one might wonder whether we need a monadic form of `gmapQ`, by analogy with `gmapT`/`gmapM`. No, we do not: a monadic *query* is just a special case of an ordinary query. To see that, we need only recognise that `Maybe` is a monad, so the `find` operation of Section 4.2 is really performing a monadic query.

6 Refinements and reflections

Having introduced the basics, we pause to reflect on the ideas a little and to make some modest generalisations.

6.1 An aside about types

It is worth noticing that the type of `everywhere` could equivalently be written thus:

```
everywhere :: (forall b. Term b => b -> b)
           -> (forall a. Term a => a -> a)
```

by moving the implicit `forall a` inwards. The nice thing about writing it this way is that it becomes clear that `everywhere` is a generic-transformation transformer. We might even write this:

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

The same approach gives a more perspicuous type for `everything`:

```
type GenericQ r = forall a. Term a => a -> r
everything :: (r -> r -> r)
           -> GerericQ r -> GerericQ r
```

From a type-theoretic point of view, these type signatures are identical to the original ones, and GHC supports such isomorphisms directly. In particular, GHC allows a `forall` in `type` synonym declarations (such as `GenericT`) and allows a `forall` to the right of a function arrow (which happens when the type synonym is expanded).

6.2 Richer traversals

Sometimes we need to combine generic queries and transformations. For example, suppose we want to increase the salaries of everyone in a named department, leaving everyone else's salary unchanged. The main function is a generic transformation, `incrOne`, but it uses the services of a generic query `isDept`:

```
incrOne :: Name -> Float -> GenericT
incrOne n k a
    | isDept n a = increase k a
    | otherwise  = gmapT (incrOne d k) a

isDept :: Name -> GenericQ Bool
isDept n = False ‘mkQ‘ isDeptD n

isDeptD :: Name -> Dept -> Bool
isDeptD n (D n’ _ _) = n==n’
```

`incrOne` first tests its argument to see whether it is the targeted department but, because `incrOne` is a *generic* transformation, it must use a generic query, `isDept` to make the test. The latter is built just as before using `mkQ`. Returning to `incrOne`, if the test returns `True`, we call `increase` (from Section 3) on the department[4]; otherwise we apply `incrOne` recursively to the children.

In this case we did not use one of our traversal combinators (`everything`, `everywhere`, etc.) to do the job; it turned out to be more convenient to write the recursion explicitly. This is yet another example of the benefit of keeping the recursion out of the definition of the `gmap` functions.

6.3 Identifying the interesting cases

Our generic programming technique encourages fine type distinctions via algebraic data types as opposed to anonymous sums and products. The specific data types usually serve for the identification of interesting cases in a generic algorithm. For example, we used a separate data type for `Salary`:

```
data Salary = S Float
```

If we had instead used an ordinary `Float` instead of `Salary`, and if the `Person` type also included a `Float` (the person's height, perhaps) the `increase` of Section 3 might end up increasing everyones height as well as their salary!

If this happens, one solution is to add more type distinctions, i.e., declarations of datatypes and newtypes as opposed to type synonyms. Another is simply to include some more context to the program in terms of the intercepted patterns. Thus, instead of using `mkT` to build special case for `Float`, build a special case for `Employee`:

```
increase k = everywhere (mkT (incE k))

incE :: Float -> Employee -> Employee
incS k (E p s) = E p (s * (1+k))
```

There is a dual problem, which is persuading the traversal functions to *stop*. The programmer might want to cut off traversal explicitly at certain kinds of nodes. In the case of a transformation, such cut-offs are useful to restrict the extent of changes in the tree. For example, we could further parameterise `everywhere` by a generic query that returns `True` if the traversal should not visit the sub-tree:

```
everywhereBut :: GenericQ Bool
              -> GenericT -> GenericT
everywhereBut q f x
  | q x       = x
  | otherwise = f (gmapT (everywhereBut q f) x)

increase k = everywhereBut names (mkT (incS k))

names :: GenericQ Bool
names = False ‘mkQ‘ isName

isName :: String -> Bool
isName n = True
```

Writing such "stop conditions" is useful not only to restrict the coverage of traversal, but also to avoid "fruitless traversal". For example, the `increase` function will unnecessarily traverse every character of the department's name, and also of each person's name. (In Haskell, a `String` is just a list of `Char`.) From the point of view of the generic function, it is entirely possible that there might be a `Salary` buried inside the name. Writing efficiency-directed stop conditions is undoubtedly tiresome, and is a shortcoming of our approach. It can only be avoided by an analysis of the datatype structure, which is certainly feasible, but only with compiler support.

6.4 Compound type extension

Continuing the same example, what if there happened to be two or more uninteresting types, that we wanted to refrain from traversing? Then we would need a generic query that returned `True` for any of those types, and `False` otherwise. Compound type extensions like this are the topic of this section.

The general question is this: given a generic query, how can we extend it with a new type-specific case? We need `extQ`, a cousin of `mkQ`:

```
extQ :: (Typeable a, Typeable b)
     => (a -> r) -> (b -> r) -> (a -> r)
(q ‘extQ‘ f) a = case cast a of
                   Just b  -> f b
                   Nothing -> q a
```

We can now build a generic query that has arbitrarily many special cases simply by composing `extQ`. There are similar type-extension functions, `extT` and `extM`, that allow a generic transformation to have an arbitrary number of type-specific cases.

Here is a more interesting example. Suppose we want to generate an association list, giving the total head-count for each department:

```
headCount :: Company -> [(Name,Int)]
headCount c = fst (hc c)

type HcInfo = ([(Name,Int)], Int)

hc :: GenericQ HcInfo
```

The main generic function, `hc`, returns an `HcInfo`; that is, a pair of the desired association list together with the total head count of the sub-tree. (Returning a pair in this way is just the standard tupling design pattern, nothing to do with generic programming.) First we define the the type-specific cases for the two types `Dept` and `Person` of interest:

```
hcD :: Dept -> [HcInfo] -> HcInfo
hcD (D d _ us) kids = ((d,n):l, n)
     where
       (l,n) = addResults kids

hcP :: Person -> [HcInfo] -> HcInfo
hcP p _ = ([], 1)
```

[4]Actually, Section 3 gave a monomorphic type to `increase`, whereas we need it to have a generic type here, so we would have to generalise its type signature.

```
addResults :: [HcInfo] -> HcInfo
addResults rs = (concat (map fst rs),
                 sum    (map snd rs))
```

Each of them takes a list of `HcInfo`, the head-count information for the child nodes (irrelevant for a `Person`), and the node itself, and builds the head-count information for the node. For a person we return a head-count of 1 and an empty list of departments; while for a department we add the department to the list of sub-departments, plus one for the manager herself. Now we can combine these functions using a new traversal combinator `queryUp`:

```
queryUp :: (forall a. Term a => a -> [r] -> r)
        -> GenericQ r
queryUp f x = f x (gmapQ (queryUp f) x)

hc :: GenericQ HcInfo
hc = queryUp (hcG ‘extQ‘ hcP ‘extQ‘ hcD)

hcG :: Term a => a -> [HcInfo] -> HcInfo
hcG node kids = addResults kids
```

Here `queryUp` first deals with the children (via the call to `gmapQ`), and then applies the specified function to the node `x` and the query results of the children. The main function, `hc`, calls `queryUp` with a function formed from a generic case `hcG`, with two type extensions for `hcP` and `hcD`. As an aside, we are using generic queries with a higher-order result type here, namely `[r]->r`.

6.5 Strange types

Programming languages like ML and Haskell permit rather free-wheeling data type definitions. Algebraic data types can be mutually recursive, parameterised (perhaps over higher-kinded type variables), and their recursion can be non-uniform. Here are some typical examples (the last one is taken from [3]):

```
data Rose a = MkR a [Rose a]
data Flip a b = Nil | Cons a (Flip b a)

data E v   = Var v | App (E v) (E v) | Lam (E (Inc v))
data Inc v = Zero | Succ v
```

For all of these the `Term` instance declaration follow the usual form. For example, here is the `Term` instance for `Rose`:

```
instance Term a => Term (Rose a) where
  gmapT f (MkR a rs) = MkR (f a) (f rs)
  gmapQ f (MkR a rs) = [f a, f rs]
  gmapM f (MkR a rs) = do a′  <- f a
                          rs′ <- f rs
                          return (MkR a′ rs′)
```

Components of algebraic data types can also involve local quantifiers and function types. The former do not necessitate any specific treatment. As for the latter, there is of course no extensional way to traverse into function values unless we meant to traverse into the source code of functions. However, encountering functions in the course of traversal does not pose any challenge. We can treat functions as atomic data types, once and for all, as shown here:

```
instance Term (a -> b) where
  gmapT f x = x
  gmapQ f x = []
  gmapM f x = return x
```

Type-safe cast copes with all these strange types as well because it is not at all sensitive to the structure of the datatype components. The `Typeable` instances deal with the *names* of the datatypes, and the names of their parameter types or type constructors.

7 Generalising `gmap`

We have seen three different maps, `gmapT`, `gmapQ`, and `gmapM`. They clearly have a lot in common, and have a rich algebra. For example:

$$
\begin{aligned}
\texttt{gmapT id} &= \texttt{id} \\
\texttt{gmapT f . gmapT g} &= \texttt{gmapT (f . g)} \\
\texttt{gmapQ f . gmapT g} &= \texttt{gmapQ (f . g)}
\end{aligned}
$$

Two obvious questions are these: (a) might a new application require a new sort of `gmap`? (b) can we capture all three as special cases of a more general combinator?

So far as (a) is concerned, any generic function must have type

$$\texttt{Term a => a ->}\, F(\texttt{a})$$

for some type-level function F. We restrict ourselves to type-polymorphic functions F; that is, F can return a result involving `a`, but cannot behave differently depending on `a`’s (type) value. Then we can see that F can be the identity function (yielding a generic transformation), ignore `a` (yielding a query), or return some compound type involving `a`. In the latter case, we view $F(\texttt{a})$ as the application of a parameterised type constructor. We covered the case of a monad via `gmapM` but we lack coverage for other type constructors. So indeed, a generic function with a type of the form

```
Term a => a -> (a,a)
```

is not expressible by any of our `gmap` functions.

But all is not lost: the answer to question (b) is “yes”. It turns out that all the generic maps we have seen are just special instances of a more fundamental scheme, namely a fold over constructor applications. At one level this comes as no surprise: from dealing with folds for lists and more arbitrary datatypes [22], it is known that mapping can be regarded as a form of folding. However, *it is absolutely not straightforward to generalise the map-is-a-fold idea to the generic setting*, because one usually expresses map as a fold by instantiating the fold’s arguments in a data-type-specific way.

In this section we show that by writing fold in a rather cunning way it is nevertheless possible to express various maps in terms of a single fold in a generic setting. Before diving in, we remark that this section need not concern the application programmer: our three `gmap`s have been carefully chosen to match a very large class of applications directly.

7.1 The generic fold

We revise the class `Term` for the last time, adding a new operator `gfoldl`. We will be able to define all three `gmap` operators using `gfoldl` but we choose to leave them as methods of the class. Doing so means that when giving an instance for `Term` the programmer may, if she wishes, define `gmapT` etc. directly, as we have done earlier in the paper.

```
class Typeable a => Term a  where
 gmapT :: (forall b. Term b => b -> b) -> a -> a
 gmapQ :: (forall b. Term b => b -> r) -> a -> [r]
 gmapM :: Monad m
       => (forall b. Term b => b -> m b) -> a -> m a
 gfoldl :: (forall a b. Term a => w (a -> b)
                               -> a -> w b)
        -> (forall g. g -> w g)
        -> a -> w a
```

Trying to understand the type of `gfoldl` directly can lead to brain damage. It is easier to see what the instances look like. Here is the instance for the types `Employee` and `SubUnit`:

```
instance Term SubUnit where
  gfoldl k z (PU p) = z PU ‘k‘ p
  gfoldl k z (DU d) = z DU ‘k‘ d
```

```
instance Term Employee where
  gfoldl k z (E p s) = (z E `k` p) `k` s
```

Notice that *the constructor itself* (E, or PU etc.) is passed to the z function as a base case; this is the key difference from a vanilla fold, and is essential to generic definitions of gmapT etc. using gfoldl. In particular:

```
gfoldl ($) id x = x
```

That is, instantiating z to the identity function, and k to function application ($) simply rebuilds the input structure. That is why we chose a left-associative fold: because it matches the left-associative structure of function application.

7.2 Using `gfoldl`

We will now show that gmapT and friends are just special instances of gfoldl. That idea is familiar from the world of lists, where map can be defined in terms of foldr. Looking at an instance helps to make the point:

```
gmapT f    (E p s) = E (f p) (f s)
gfoldl k z (E p s) = (z E `k` p) `k` s
```

How can we instantiate k and z so that gfoldl will behave like gmapT? We need z to be the identity function, while k should be defined to apply f to its second argument, and then apply its first argument to the result:

```
gmapT f = gfoldl k id
        where
          k c x = c (f x)
```

Operationally this is perfect, but the types are not quite right. gmapT returns a value of type a while gfoldl returns a (w a). We would like to instantiate w to the identity function (at the type level), obtaining the following specialised type for gfoldl:

```
gfoldl :: (forall a b. Term a => (a -> b)
                                -> a -> b)
       -> (forall g. g -> g)
       -> a -> a
```

However, functions at the type level make type inference much harder, and in particular, Haskell does not have them. The solution is to instantiate w to the type constructor ID accompanied by some wrapping and unwrapping:

```
newtype ID x = ID x

unID :: ID a -> a
unID (ID x) = x

gmapT f x = unID (gfoldl k ID x)
  where
    k (ID c) x = ID (c (f x))
```

The ID constructor, and its deconstructor unID are operationally no-ops, but they serve to tell the type checker what to do. The encoding of gmapM is very similar to the one for gmapT. We use do notation instead of nested function application. The type of gmapM does not require any wrapping because the monad type constructor directly serves for the parameter w. That is:

```
gmapM f = gfoldl k return
  where
    k c x = do c' <- c
               x' <- f x
               return (c' x')
```

The last one, gmapQ, is a little more tricky because the structure processed by gfoldl is left-associative, whereas the structure of the list returned by gmapQ is right-associative. For example:

```
gmapQ f    (E p s) = f p : (f s : [])
gfoldl k z (E p s) = (z E `k` p) `k` s
```

There is a standard way to solve this, using higher-order functions:

```
gmapQ f = gfoldl k (const id) []
  where
    k c x rs = c (f x : rs)
```

However, again we must do some tiresome type-wrapping to explain to the type inference engine why this definition is OK:

```
newtype Q r a = Q ([r]->[r])
unQ (Q f) = f

gmapQ f x = unQ (gfoldl k (const (Q id)) x) []
  where
    k (Q c) x = Q (\rs -> c (f x : rs))
```

Notice that (Q r) is a constant function at the type level; it ignores its second parameter a. Why? Because a query returns a type that is independent of the type of the argument data structure.

7.3 Summary

We contend that one-layer folding is the fundamental way to perform term traversal in our framework. This section has shown that the gmap functions can all be defined in terms of a single function gfoldl. Lest the involved type-wrapping seems onerous, we note that it occurs *only* in the definitions of the gmap functions in terms of gfoldl. The programmer need never encounter it. The gmap definitions in terms of gfoldl might not be very efficient because they involve some additional amount of higher-order functions. So the programmer or the implementor of the language extension has a choice. Either the gmap operators are defined directly per datatype, or they are defined in terms of gfoldl once and for all via the shown "default" declarations.

8 Type-safe cast

Our entire approach is predicated on the availability of a type-safe cast operator, which in turn is closely related to intensional polymorphism and dynamic typing. We will discuss such related work in Section 9.3. In fact, it is well-known folk lore in the Haskell community that much of the functionality of cast can be programmed in standard Haskell. Strangely, there is no published description of this trick, so we review it here, giving an encoding that can be regarded as a reference implementation.

8.1 The `Typeable` class

The key idea is to refine the type class Typeable, which was previously assumed to be abstract, as follows:

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

The overloaded operation typeOf takes a value and returns a runtime representation of its type. Here is one possible implementation of the TypeRep type, and some instances:

```
data TypeRep = TR String [TypeRep]

instance Typeable Int where
  typeOf x = TR "Prelude.Int" []

instance Typeable Bool where
  typeOf x = TR "Prelude.Bool" []

instance Typeable a => Typeable [a] where
  typeOf x = TR "Prelude.List" [typeOf (get x)]
           where
             get :: [a] -> a
             get = undefined

instance (Typeable a, Typeable b)
      => Typeable (a->b) where
  typeOf f = TR "Prelude.->" [typeOf (getArg f),
                              typeOf (getRes f)]
           where
```

```
getArg :: (a->b) -> a
getArg = undefined

getRes :: (a->b) -> b
getRes = undefined
```

Notice that `typeOf` *never evaluates its argument*. In particular, the call `(get x)` in the list instance will never be evaluated[5]; it simply serves as a proxy, telling the compiler the type at which to instantiate the recursive call of `typeOf`, namely to the element type of the list. If Haskell had explicit type arguments, `typeOf` could dispense with its value argument, with its calls using type application alone.[6]

8.2 Defining `cast` using `typeOf`

Type-safe cast is easy to implement given `typeOf`, plus a small Haskell extension:

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
cast x = r
       where
         r = if typeOf x == typeOf (get r)
             then Just (unsafeCoerce x)
             else Nothing

         get :: Maybe a -> a
         get x = undefined
```

Here we check whether the argument `x` and result `r` have the same type representation, and if so coerce the one into the other. Here, `unsafeCoerce` is an extension to Haskell, with the following type:

```
unsafeCoerce :: a -> b
```

It is easy to implement: operationally it is just the identity function. It is, of course, just as unsafe as its name implies, and we do not advocate its wide-spread use. Rather, we regard `unsafeCoerce` as an implementation device to implement a safe feature (`cast`); many language implementations contain a similar trap-door.

8.3 What a mess?

At this point the reader may be inclined to throw up his hands and declare that if this paper requires `unsafeCoerce`, and `instance` declarations with magic strings that must be distinct, then it has no place in a language like Haskell. But note that the above scheme is meant by us as a reference implementation as opposed to a programming technique.

That is, the compiler should provide direct support for the class `Typeable`, so that its instance for each data type is automatically generated by the compiler. The programmer does not instantiate the class him- or herself. Furthermore, `cast` should be provided as a primitive — it may be *implemented* inside the system library with some kind of low-level coercion, but that is invisible to (and inaccessible to) the application programmer. With this degree of compiler support, the system is indeed type-safe.

So this section does not present a programming technique for the user. Rather, it shows that compiler support for `cast` does not require some mysterious fiddling with runtime data representations. Instead, somewhat surprisingly, it can be cleanly implemented using Haskell's type-class framework with some readily-generated simple instance declarations. Furthermore, albeit as an unsavoury stop-gap measure, it is a real advantage to be able to prototype the system without requiring any compiler support except `unsafeCoerce`.

[5] The value `undefined` has type `forall a.a` in Haskell.

[6] GHC supports scoped type variables, so a nicer way to write the list instance of `typeOf` is this:

```
TR "Prelude.List" [typeOf (undefined :: a)]
```

One might worry about efficiency, because `cast` involves comparing `TypeRep` data structures. That cost, however, is not fundamental. The `TypeRep` structures can readily be hash-consed (especially if there is direct compiler support) so that they can be compared in constant time. Again, this is the business of the library writer (or even compiler implementor) not the application programmer.

9 Alternative approaches

Generic programming has received a great deal of attention, and we review the work of others in this section. Before we do, it is worth mentioning that one very brutal approach to generic programming lies readily to hand, namely *using a universal data type*, such as:

```
data Univ = I Int | S String | ... etc. ...
          | B ConstrName [Univ]
type ConstrName = String
```

A generic program works by (a) converting (embedding) the input data structure to `Univ`, (b) traversing the universal data structure, and (c) converting (projecting) the result back to the original type. This approach has the merit of simplicity, but it is inefficient, and (worse) completely untyped. In step (b) there is no static check that, when matching on a constructor named `"Person"`, the correct number or type of fields are matched. There are ways to improve the type safety and efficiency of this approach; for example, one can use an abstract datatype for generic functions to separate typed and untyped code [17]. However, we concentrate on statically-typed approaches in the rest of this section.

9.1 Rank-2 types

The Hindley-Milner type system is gracefully balanced on a cusp between expressiveness and decidability. A polymorphic type may be quantified only at the outermost level — this is called a *rank-1 type* — but in exchange a type inference engine can find the most general type for any typeable program, without the aid of any type annotations whatsoever.

Nevertheless, higher-ranked types are occasionally useful. A good example is the type of `build`, the list-production combinator that is central to the short-cut deforestation technique [6]. Its type is:

```
build :: forall a. (forall b. (a->b->b) -> b -> b)
                -> [a]
```

Another example is `runST`, the combinator that encapsulates a stateful computation in a pure function [19]:

```
runST :: forall a. (forall s. ST s a) -> a
```

It is well known that type inference for programs that use higher-ranked types is intractable [16]. Nevertheless, it is not only tractable but easy if sufficient type annotations are given [24]. The two Haskell implementations GHC and Hugs support data constructors with rank-2 types; the type inference problem is easier here because the data constructor itself acts as a type annotation. However that would be very inconvenient here: `gmapT` is not a data constructor, and it would require tiresome unwrapping to make it so.

So in fact GHC uses a type inference algorithm that permits any function to have a type of arbitrary-rank type, provided sufficient type annotations are given. The details are beyond the scope of this paper, but are given in [31]. We believe that the `gmap` family of functions offers further evidence of the usefulness of rank-2 types in practical programming.

9.2 Generic traversal

Polytypic programming

The core idea underlying polytypic programming [15, 14, 10] is to define a generic function by induction on the structure of an argument type or the result type of a function. Induction is usually supported by a corresponding language extension: the function definition has cases for sums, products, and others. This approach initially leads to *purely*-generic functions; that is, ones driven entirely by the structure of the type. Examples include serialisation and its inverse, comparison operations, and hashing. Unfortunately, these are just about the *only* purely-generic operations, and our own view is that purely-generic programming is too restrictive to be useful.

Thus motivated, customisation of generic programs is addressed in the Generic Haskell program. In [4], techniques are discussed to extend a polytypic function with cases for a particular constructor or a type. Generic Haskell is a very substantial extension to Haskell, whereas our proposal is much more lightweight and better integrated with ordinary functional programming. Furthermore, in Generic Haskell, a generic function is not a first-class citizen. That is, one cannot write generic functions operating on other generic functions, as our traversal combinators (e.g., `everywhere`) require. Also, (run-time and nominal) type-safe cast is alien to polytypic programming. Using techniques such as those in [4], one can *encode* traversals as opposed to using our combinator style.

Derivable type classes [11] is another extension of Haskell to support generic programming. The idea here is that a generic function is just a template that specifies how to generate an `instance` declaration for the generic function for each data type. It is easy to over-ride this template for specific types. Again, derivable type-classes are oriented towards structural induction (not nominal analysis) over types; recursion is built into each generic function; and each new generic function requires a new class. Derivable type classes (combined with rank-2 types) are sufficient to define the `gmap` family of functions or the `gfoldl` function, with a modest amount of encoding. However, derivable type classes are not suitable to define our nominal type case because of their bias towards structural induction.

Generalised folds

It is a well-established idea that maps and folds can be defined for all kinds of datatypes, even for systems of datatypes [22, 29, 23]. The inherent assumption is here that recursion into compound terms is performed *by the fold operation itself*. This sets this idea apart from our simpler and yet more general approach where layer-wise traversal is favoured. This way, we allow the programmer to wire up recursion in any way that is found convenient. Besides the anticipated recursion, generalised folds suffer from another problem articulated in [18]: if larger systems of datatypes are considered, it is impractical to enumerate all the ingredients for folding by hand. In effect, this is another instance of boilerplate: most ingredients follow a certain scheme, only few ingredients should be provided by the programmer. To this end, *updatable* fold algebras were proposed in [18]. The present development generalises (updatable) generalised folds in several dimensions. Firstly, type extension can operate at the type level whereas fold algebras are updated at the constructor level. Secondly, generic traversal allows to define all kinds of traversal schemes as opposed to simple catamorphic or paramorphic fold. Thirdly, the fold algebra approach suffers from a closed-world assumption; adding new data types is not straightforward. No such assumption is present in our present development.

The non-recursive map trick

The non-recursive map trick (introduced in Sections 3.2 and 3.3) has been known in the functional programming community for some time, e.g., in the sense of programming with functors [22, 28]. In this approach, for every recursive data type, `Tree` say, one defines an auxiliary type, `Tree′` that is the functor for `Tree`:

```
data Tree a    = Leaf a | Fork (Tree a) (Tree a)
data Tree′ t a = Leaf′ a | Fork′ t t
```

Now the following type isomorphism holds:

$$\texttt{Tree}'\ \texttt{(Tree a) a} \equiv \texttt{Tree a}$$

Recursive traversals can then be defined as recursive functions in terms of a one-layer functorial `map`. To use this approach directly for practical programming, one needs to write functions to convert to and from between these the above isomorphic types, and the situation becomes noticeably more complicated when there are many mutually-recursive types involved [32, 28], and breaks down altogether when the recursion is non-uniform [25]:

```
data Seq a = Nil | Cons a Seq (a,a)
```

In contrast, our approach does not require an auxiliary data type, works fine for arbitrary datatypes. and it also copes with systems of mutually recursive datatypes. This is a major improvement over previous work, and this makes the technique more likely to be used in practice. In an untyped setting, the idea to map over the immediate children of a term is rather straightforward, e.g., in Prolog. Indeed, it seems that a very similar technique has been used in Lisp community already for quite some time.[7]

The idea of building a library of combinators that facilitate first-class tree-traversal *strategies* (e.g. top-down, bottom-up, repeat-until, leftmost-first, etc.) in terms of one-layer traversal steps is also well established in the term-rewriting community. This idea has seen a flurry of recent activity. There are three main approaches to the combinator style. One is to define a new language for strategic programming. A prime example is the untyped language Stratego [33]. Another approach that can also be used to support strategies in an existing functional language is to transform the input data into a single universal data type, and write generic strategies over that universal data type; a good example is the HaXML combinator library [35]. Yet another approach that works particularly well with functional programming is to model strategies as abstract datatypes. The implementations of the strategy combinators can then hide some encoding needed to present "strategies as functions" to the programmer. This approach underlies the *Strafunski* programme.[8] All these streams of work describe a rich library of strategy combinators. Our new contribution is *to show how the strategic-combinator approach to traversal can be smoothly accommodated in a typed functional language, where term traversals are ordinary functions on the user-defined data types*. Also, the employment of rank-2 types and the identification of the fundamental folding operator improves on the encodings and combinator suites in previous work.

The visitor pattern

In object-oriented programming, the *visitor pattern* is the classic incarnation of recursive traversal. In fact, though, an instance of the visitor pattern is rather like the problematic `increase` that we started with in Section 2: the visitor requires a case for each data type (say, class), and the traversal is mixed up with the processing to be done to each node [26]. Many variations on the basic visitor pattern have been proposed. Palsberg suggests a more generic alternative, the `Walkabout` class, based on reflection; its performance is poor, and Palsberg offers an interesting discussion of other design choices [26]. A generative approach to flexible support for programming with visitors is suggested by Visser [34] accompanied with a discussion of other generative approaches. Given a class hierarchy, an interface for visitor combinators is instantiated very much in the style of strategic programming (see above). Node processing and recursive traversal is effectively separated, and arbitrary traversal schemes can be defined.

---

[7] Personal communication Alex Aiken.

[8] `http://www.cs.vu.nl/Strafunski`

Lieberherr's et al.'s *adaptive programming* offers a high-level approach to traversal of object structures [21] when compared to visitors. This style assumes primitives to specify pieces of computation to be performed along paths that are constrained by starting nodes, nodes to be passed, nodes to be by-passed, and nodes to be reached. Adaptive programs are typically implemented by a language extension, a reflection-based API, or by compilation to a visitor.

9.3 Type-safe cast

There are two main ways to implement type-safe cast, each with an extensive literature: intensional type analysis; or dynamic typing.

Intensional type analysis enables one to write functions that depend on the (run-time) type of a value [8, 37]. To this end, one uses a `typecase` construct to examine the actual structure of a type parameter or the type of a polymorphic entity, with `case` alternatives for sums, products, function types, and basic datatypes. This structural type analysis can also be performed recursively (as opposed to mere one-level type case). Checking for type equality is a standard example, and so looks like a promising base for a type-safe cast, as Weirich shows [36].

There are two difficulties. First, adding intensional polymorphism to the language is a highly non-trivial step. Second, and even more seriously, all the work on intensional polymorphism is geared towards *structural* type analysis, whereas our setting absolutely requires *nominal* type analysis (cf. [7]). For example, these two types are structurally equal, but not nominally equal:

```
data Person = P String Float  -- Name and height
data Dog    = D String Float  -- Name and weight
```

We should not treat a `Person` like a `Dog` — or at least we should allow them to be distinguished.

There is a great deal of excellent research on introducing *dynamic types* into a statically-typed language; for example [1, 2, 20]. However, it addresses a more general question than we do, and is therefore much more complicated than necessary for our purpose. In particular, we do not need the type `Dynamic`, which is central to dynamic-typing systems, and hence we do not need `typecase` either, the principal language construct underlying dynamic typing.

The class `Typeable` and the `unsafeCoerce` function, are the foundation of the `Dynamic` library, which has been a standard part of the Hugs and GHC distributions for several years. However, it seems that the material of Section 8 has never appeared in print. The key idea first appeared in an 1990 email from one of the current authors to the (closed) `fplanc` mailing list [27], later forwarded to the (open) Haskell mailing list [12]. The `cast` function is not so well known, however; the first reference we can trace was a message to the Haskell mailing list from Henderson [9].

10 Concluding remarks

We have presented a practical design pattern for generic programming in a typed functional setting. This pattern encourages the programmer to avoid the implementation of tiresome and maintenance-intensive boilerplate code that is typically needed to recurse into complex data structures. This pattern is relevant for XML document processing, language implementation, software reverse and re-engineering. Our approach is simple to understand because it only involves two designated concepts of one-layer traversal and type cast. Our approach is general because it does not restrict the datatypes subject to traversal, and it allows to define arbitrary traversal schemes — reusable ones but also application-specific ones. Language support for the design pattern was shown to be simple. The approach takes advantage of research to put rank-2 type systems to work.

Performance

Our benchmarks show that generic programs are reasonably efficient (see also the accompanying software distribution). The generic program for salary increase, for example, is 3.5 times slower [9] than the normal hand-coded program. The dominant cause of this penalty is our sub-optimal encoding technique for type-safe cast. Recall that generic traversals perform a comparison of type representations for every encountered node at run-time. So it is crucial to make type representations very efficient, preferably via built-in support. A hand-written solution does not involve any such checks. The above factor is also caused by the fact that generic traversal schemes are not accessible to a number of optimisations which are available for hard-wired solutions. This is because the `gmap` family relies on the `Term` class and higher-order style. Finally, recall that generic traversals tend to traverse more nodes than necessary if extra precautions are omitted to stop recursion.

Perspective

We are currently investigating options to support the key combinators `cast` and `gfoldl` (or the `gmap` family) efficiently by the GHC compiler for Haskell. Such a native implementation will remove the penalty related to the comparison of type representations, and it will render external generative tool support unnecessary. As the paper discusses, such built-in support is not hard to provide, but there is some design space to explore. We are also working on automating the derivation of stop conditions for traversals based on reachability properties of the recursive traversal schemes and the traversed data structure. We envisage that a template-based approach [30] can be used to derive optimised traversals at compile time.

Acknowledgements

We thank Nick Benton, Robert Ennals, Barry Jay, Johan Jeuring, Ralf Hinze, Tony Hoare, Simon Marlow, Riccardo Pucella, Colin Runciman, Joost Visser, and Stephanie Weirich for very helpful discussions and feedback on earlier drafts of this paper.

11 References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *16th ACM Conference on Principles of Programming Languages*, pages 213–227, Jan. 1989.

[2] M. Abadi, L. Cardelli, B. Pierce, and D. Remy. Dynamic typing in polymorphic languages. In *Proceedings of the 1992 ACM Workshop on ML and its Applications*, pages 92–103, San Francisco, June 1992.

[3] R. Bird and R. Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, Jan. 1999.

[4] 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, 2002. To appear.

[5] *ACM Conference on Functional Programming and Computer Architecture (FPCA'93)*, Cophenhagen, 1993. ACM. ISBN 0-89791-595-X.

[6] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In FPCA93 [5], pages 223–232. ISBN 0-89791-595-X.

[9] Test environment: Linux-i386, Pentium III, 512 MB, 256 KB cache, Thinkpad A22p, GHC 5.04 with optimisation package enabled.

[7] N. Glew. Type dispatch for named hierarchical types. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34.9 of *ACM Sigplan Notices*, pages 172–182, N.Y., Sept. 27–29 1999. ACM Press.

[8] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 130–141. ACM, Jan. 1995.

[9] F. Henderson. Dynamic type class casts proposal. Email to the `haskell` mailing list, Oct. 1999.

[10] R. Hinze. A New Approach to Generic Functional Programming. In T. Reps, editor, *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, January 19-21*, pages 119–132, Jan. 2000.

[11] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proceedings of the 2000 Haskell Workshop, Montreal*, number NOTTCS-TR-00-1 in Technical Reports, Sept. 2000.

[12] P. Hudak. Phil's proposal for restricted type classes. Email to the `haskell` mailing list, June 1991.

[13] R. Hughes, editor. *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, volume 523 of *Lecture Notes in Computer Science*, Boston, 1991. Springer Verlag.

[14] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 470–482, Paris, Jan. 1997. ACM.

[15] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *2nd Int. School on Advanced Functional Programming, Olympia, WA, USA, 26–30 Aug 1996*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, Berlin, 1996.

[16] A. Kfoury. Type reconstruction in finite rank fragments of second-order lambda calculus. *Information and Computation*, 98(2):228–257, June 1992.

[17] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proc. Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, Jan. 2002.

[18] R. Lämmel, J. Visser, and J. Kort. Dealing with Large Bananas. In J. Jeuring, editor, *Proceedings of WGP'2000, Technical Report, Universiteit Utrecht*, pages 46–59, July 2000.

[19] J. Launchbury and S. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–342, Dec. 1995.

[20] X. Leroy and M. Mauny. Dynamics in ML. In Hughes [13].

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

[22] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In Hughes [13], pages 124–144.

[23] E. Meijer and J. Jeuring. Merging Monads and Folds for Functional Programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 228–266. Springer Verlag, 1995.

[24] M. Odersky and K. Läufer. Putting type annotations to work. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 54–67. ACM, St Petersburg Beach, Florida, Jan. 1996.

[25] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1998.

[26] J. Palsberg and B. Jay. The essence of the visitor pattern. In *Proceedings 22nd Annual International Computer Software and Applications Conference (COMPSAC'98)*, pages 9–15, Aug. 1998.

[27] S. Peyton Jones. Restricted overloading. Email to the `fplangc` mailing list, Dec. 1990.

[28] T. Sheard. Generic unification via Two-Level types and parameterized modules. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, volume 36, 10 of *ACM SIGPLAN notices*, pages 86–97, Florence, Sept. 2001. ACM.

[29] T. Sheard and L. Fegaras. A fold for all seasons. In FPCA93 [5], pages 233–242. ISBN 0-89791-595-X.

[30] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In M. Chakravarty, editor, *Proceedings of the 2002 Haskell Workshop, Pittsburgh*, Oct. 2002.

[31] M. Shields and S. Peyton Jones. Putting "putting type annotations to work" to work. In preparation, 2002.

[32] S. Swierstra, P. Alcocer, and J. Saraiva. Designing and implementing combinator languages. In S. Swierstra, P. Henriques, and J. Oliveira, editors, *Advanced Functional Programming, Third International School, AFP '98*, volume 1608 of *Lecture Notes in Computer Science*, pages 150–206, Braga, Portugal, Sept. 1999. Springer Verlag.

[33] E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 13–26, Baltimore, 1998. ACM.

[34] J. Visser. Visitor combination and traversal control. In *OOPSLA 2001 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 2001.

[35] 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.

[36] S. Weirich. Type-safe cast. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 58–67, Montreal, Sept. 2000. ACM.

[37] S. Weirich. Higher-order intensional type analysis. In D. L. Métayer, editor, *Programming Languages and Systems: 11th European Symposium on Programming (ESOP 2002), Grenoble, France*, number 2305 in Lecture Notes in Computer Science, pages 98–114. Springer Verlag, 2002.

[38] N. Winstanley. A type-sensitive preprocessor for Haskell. In *Glasgow Workshop on Functional Programming, Ullapool*, 1997.