Prototyping Generic Programming in Template Haskell

Ulf Norell and Patrik Jansson*

Computing Science, Chalmers University of Technology, Göteborg, Sweden

Abstract. Generic Programming deals with the construction of programs that can be applied to many different datatypes. This is achieved by parameterizing the generic programs by the structure of the datatypes on which they are to be applied. Programs that can be defined generically range from simple map functions through pretty printers to complex XML tools.
The design space of generic programming languages is largely unexplored, partly due to the time and effort required to implement such a language. In this paper we show how to write flexible prototype implementations of two existing generic programming languages, PolyP and Generic Haskell, using Template Haskell, an extension to Haskell that enables compile-time meta-programming. In doing this we also gain a better understanding of the differences and similarities between the two languages.

1 Introduction

Generic functional programming [9] aims to ease the burden of the programmer by allowing common functions to be defined once and for all, instead of once for each datatype. Classic examples are small functions like maps and folds [8], but also more complex functions, like parsers and pretty printers [11] and tools for editing and compressing XML documents [5], can be defined generically. There exist a number of languages for writing generic functional programs [1,3, 6,7,10,12,13], each of which has its strengths and weaknesses, and researchers in generic programming are still searching for *The Right Way*. Implementing a generic programming language is no small task, which makes it cumbersome to experiment with new designs.

In this paper we show how to use Template Haskell [16] to implement two generic programming extensions to Haskell: PolyP [10,15] and Generic Haskell [6]. With this approach, generic functions are written in Haskell (with the Template Haskell extension), so there is no need for an external tool. Furthermore the support for code generation and manipulation in Template Haskell greatly simplifies the compilation of generic functions, thus making the implementations

* This work is partially funded by the Swedish Foundation for Strategic Research as part of the research program "Cover - Combining Verification Methods in Software Development".

very lightweight and easy to experiment with. Disadvantages of this approach are that we do not get the nice syntax we can get with a custom made parser and because we have not implemented a type system, generic functions are only type checked at instantiation time.

The rest of this section gives a brief introduction to Template Haskell, PolyP and Generic Haskell (GH). Section 2 compares PolyP and GH. Section 3 introduces the concepts involved in implementing generic programming using Template Haskell. Sections 4 and 5 outline the implementations of PolyP and GH and Section 6 points to possible future work.

1.1 Template Haskell

Template Haskell [16] is a language extension implemented in the Glasgow Haskell Compiler that enables compile-time meta-programming. This means that we can define code generating functions that are run at compile-time. In short you can *splice* abstract syntax into your program using the `$(...)` notation and *lift* an expression to the abstract syntax level using the quasi-quotes `[|...|]`. Splices and quasi-quotes can be nested arbitrarily deep. For example, it is possible to define the `printf` function with the following type:

```
printf :: String -> Q Exp
```

Here `printf` takes the format string as an argument and produces the abstract syntax for the `printf` function specialized to that particular format string. To use this function we can write, for instance

```
Main> $(printf "Hello %s, number %d!") "World" 100
"Hello World, number 100!"
```

Template Haskell comes with libraries for manipulating the abstract syntax of Haskell. The result type `Q Exp` of the `printf` function models the abstract syntax of an expression. The type constructor `Q` is the quotation monad, that takes care of, for instance, fresh name generation and the `Exp` type is a normal Haskell datatype modeling Haskell expressions. Similar types exist for declarations (`Dec`) and types (`Type`). The quotation monad is built on top of the IO monad, so if we want to escape it we have to use the function `unsafePerformIO :: IO a -> a`.

The definition of `printf` might look a bit complicated with all the lifts and splices, but ignoring those we have precisely what we would have written in an untyped language.

```
printf :: String -> Q Exp
printf fmt = prAcc fmt [| "" |]
  where
    prAcc :: String -> Q Exp -> Q Exp
    prAcc fmt r = case fmt of
      '%':'d':f -> [| \n -> $(prAcc f [| $r ++ show n |]) |]
      '%':'s':f -> [| \s -> $(prAcc f [| $r ++ s      |]) |]
      c:f       -> prAcc f [| $r ++ [c] |]
      ""        -> r
```

The prAcc function uses an accumulating parameter r containing (the abstract syntax of) an expression representing the string created so far. Every time we see a % code we add a lambda at the top level and update the parameter with the argument.

The keen observer will note that this definition of printf is quadratic in the length of the format string. This is easy to fix but for the sake of brevity we chose the inefficient version, which is slightly shorter.

Template Haskell supports program reflection or reification, which means that it is possible to get hold of the type of a named function or the declaration that defines a particular entity. For example:

```
reifyType id    :: Q Type
reifyDecl Maybe :: Q Dec
```

We can use this feature to find the definitions of the datatypes that a generic function is applied to.

1.2 PolyP

PolyP [10, 15] is a language extension to Haskell for generic programming, that allows generic functions over unary regular datatypes. A regular datatype is a datatype with no function spaces, no mutual recursion and no nested recursion[1]. Examples of unary regular datatypes are [], Maybe and Rose:

```
data Rose a = Fork a [Rose a]
```

Generic programming in PolyP is based on the notion of *pattern functors*. Each datatype is associated with a pattern functor that describes the structure of the datatype. The different pattern functors are shown in Figure 1. The (:+:) pattern functor is used to model multiple constructors, (:*:) and Empty model the list of arguments to the constructors, Par is a reference to the parameter type, Rec is a recursive call, (:@:) models an application of another regular datatype and Const is a constant type. The pattern functors of the datatypes mentioned above are (the comments show the expanded definitions applied to two type variables p and r):

```
type ListF  = Empty :+: (Par :*: Rec) -- Either () (p,r)
type MaybeF = Empty :+: Par           -- Either () p
type RoseF  = Par :*: ([] :@: Rec)    -- (p, [r])
```

PolyP provides two functions inn and out to fold and unfold the top-level structure of a datatype. Informally, for any regular datatype D with pattern functor F, inn and out have the following types:

```
inn :: F a (D a) -> D a
out :: D a -> F a (D a)
```

[1] The recursive calls must have the same form as the left hand side of the definition.

```
type (g :+: h) p r = Either (g p r) (h p r)
type (g :*: h) p r = (g p r, h p r)
type Empty     p r = ()
type Par       p r = p
type Rec       p r = r
type (d :@: g) p r = d (g p r)
type Const t   p r = t
```

Fig. 1. Pattern functors

Note that only the top-level structure is folded/unfolded.

A special construct, `polytypic`, is used to define generic functions over pattern functors by pattern matching on the functor structure. As an example, the definition of `fmap2`, a generic map function over pattern functors, is shown in Figure 2. Together with `inn` and `out` these polytypic functions can be used to define generic functions over regular datatypes. For instance:

```
pmap :: (a -> b) -> D a -> D b
pmap f = inn . fmap2 f (pmap f) . out
```

The same polytypic function can be used to create several different generic functions. We can, for instance, use `fmap2` to define generic cata- and anamorphisms (generalized folds and unfolds):

```
cata :: (F a b -> b) -> D a -> b
cata f = f . fmap2 id (cata f) . out

ana :: (b -> F a b) -> b -> D a
ana f = inn . fmap2 id (ana f) . f
```

1.3 Generic Haskell

Generic Haskell [2] is an extension to Haskell that allows generic functions over datatypes of arbitrary kinds. Hinze [4] observed that the type of a generic function depends on the kind of the datatype it is applied to, hence each generic function in Generic Haskell comes with a generic (kind indexed) type. The kind indexed type associated with the generic map function is defined as follows:

```
Map {[ * ]}      s t = s -> t
Map {[ k -> l ]} s t = forall a b. Map {[ k ]} a b ->
                                   Map {[ l ]} (s a) (t b)
```

Generic Haskell uses the funny brackets (`{[ ]}`) to enclose kind arguments. The type of the generic map function `gmap` applied to a type `t` of kind `k` can be expressed as

```
gmap {| t :: k |} :: Map {[ k ]} t t
```

```
polytypic fmap2 :: (a -> c) -> (b -> d) -> f a b -> f c d
    = \p r -> case f of
          g :+: h -> fmap2 p r -+- fmap2 p r
          g :*: h -> fmap2 p r -*- fmap2 p r
          Empty   -> const ()
          Par     -> p
          Rec     -> r
          d :@: g -> pmap (fmap2 p r)
          Const t -> id

(-+-) :: (a -> c) -> (b -> d) -> Either a b -> Either c d
(f -+- g) (Left  x) = Left  (f x)
(f -+- g) (Right y) = Right (g y)

(-*-) :: (a -> c) -> (b -> d) -> (a,b) -> (c,d)
(f -*- g) (x,y) = (f x, g y)
```

Fig. 2. The definition of `fmap2` in PolyP

The second type of funny brackets (`{| |}`) encloses type arguments. Following are the types of `gmap` for some standard datatypes.

```
gmap {| Bool |}   :: Bool -> Bool
gmap {| [] |}     :: forall a b. (a -> b) -> [a] -> [b]
gmap {| Either |} :: forall a b. (a -> b) ->
                     forall c d. (c -> d) ->
                     Either a c -> Either b d
```

The kind indexed types follow the same pattern for all generic functions. A generic function applied to a type of kind $\kappa \to \nu$ is a function that takes a generic function for types of kind κ and produces a generic function for the target type of kind ν.

The generic functions in Generic Haskell are defined by pattern matching on the top-level structure of the type argument. Figure 3 shows the definition of the generic map function `gmap`. The structure combinators are similar to those in PolyP. Sums and products are encoded by `:+:` and `:*:` and the empty product is called `Unit`. A difference from PolyP is that constructors and record labels are represented by the structure combinators `Con c` and `Label l`. The arguments (`c` and `l`) contain information such as the name and fixity of the constructor or label. A generic function must also contain cases for primitive types such as `Int`. The type of the right hand side of each clause is the type of the generic function instantiated with the structure type on the left. The definitions of the structure types are shown in Figure 4. Note that the arguments to `Con` and `Label` containing the name and fixity information are only visible in the pattern matching and not in the actual types.

Generic Haskell contains many features that we do not cover here, such as type indexed types, generic abstraction and constructor cases.

```
gmap {| t :: k |} :: Map {[ k ]} t t
gmap {| :+: |} gmapA gmapB (Inl a)   = Inl (gmapA a)
gmap {| :+: |} gmapA gmapB (Inr b)   = Inr (gmapB b)
gmap {| :*: |} gmapA gmapB (a :*: b) = gmapA a :*: gmapB b
gmap {| Unit |}                      = id
gmap {| Con c |}   gmapA (Con a)     = Con   (gmapA a)
gmap {| Label l |} gmapA (Label a)   = Label (gmapA a)
gmap {| Int |}                       = id
```

Fig. 3. A generic map function in Generic Haskell

```
data a :+: b = Inl a | Inr b
data a :*: b = a :*: b
data Unit    = Unit
data Con a   = Con a
data Label a = Label a
```

Fig. 4. Structure types in Generic Haskell

2 Comparing PolyP and Generic Haskell

The most notable difference between PolyP and Generic Haskell is the set of datatypes available for generic programmers. In PolyP generic functions can only be defined over unary regular datatypes, while Generic Haskell allows generic functions over (potentially non-regular) datatypes of arbitrary kinds. There is a trade-off here, in that more datatypes means fewer generic functions. In PolyP it is possible to define generic folds and unfolds such as `cata` and `ana` that cannot be defined in Generic Haskell.

Even if PolyP and Generic Haskell may seem very different, their approaches to generic programming are very similar. In both languages generic functions are defined, not over the datatypes themselves, but over a structure type acquired by unfolding the top-level structure of the datatype. The structure types in PolyP and Generic Haskell are very similar. The differences are that in PolyP constructors and labels are not recorded explicitly in the structure type and the structure type is parameterized over recursive occurrences of the datatype. This is made possible by only allowing regular datatypes. For instance, the structure of the list datatype in the two languages is (with Generic Haskell's sums and products translated into `Either`, `(,)` and `()`):

```
type ListF a r = Either      ()        (a, r )  -- PolyP
type ListS a   = Either (Con ()) (Con (a,[a])) -- GH
```

To transform a generic function over a structure type into a generic function over the actual datatype, conversion functions between the datatype and the structure type are needed. In PolyP they are called `inn` and `out` (described in Section 1.2) and they are primitives in the language. In Generic Haskell this conversion is done by the compiler and the conversion functions are not available to the programmer.

As mentioned above, generic functions in both languages are primarily defined over the structure types. This is done by pattern matching on a type code, representing the structure of the datatype. The type codes differ between the languages, because they model different sets of datatypes, but the generic functions are defined in very much the same way. The most significant difference is that in Generic Haskell the translations of type abstraction, type application and type variables are fixed and cannot be changed by the programmer.

Given a generic function over a structure type it should be possible to construct a generic function over the corresponding datatype. In Generic Haskell this process is fully automated and hidden from the programmer. In PolyP, however, it is the programmer's responsibility to take care of this. One reason for this is that the structure types are more flexible in PolyP, since they are parameterized over the recursive occurrences of the datatype. This means that there is not a unique datatype generic function for each structure type generic function. For instance the structure type generic function `fmap2` from Figure 2 can be used not only to define the generic map function, `pmap`, but also the generic cata- and anamorphisms, `cata` and `ana`.

3 Generic Programming in Template Haskell

Generic functions in both PolyP and Generic Haskell are defined by pattern matching over the code for a datatype. Such a generic function can be viewed as an algorithm for constructing a Haskell function given a datatype code. For instance, given the type code for the list datatype a generic map function can generate the definition of a map function over lists. Program constructing algorithms like this can be implemented nicely in Template Haskell; a generic function is simply a function from a type code to the abstract syntax for the function specialized to the corresponding type. When embedding a generic programming language like PolyP or Generic Haskell in Template Haskell there are a few things to consider:

- **Datatype codes**
  The structure of a datatype has to be coded in a suitable way. How this is done depends, of course, on the set of datatypes to be represented, but we also have to take into account how the type codes affect the generic function definitions. Since we are going to pattern match on the type codes we want them to be as simple as possible.
  To avoid having to create the datatype codes by hand we can define a (partial) function from the abstract syntax of a datatype definition to a type code. The (abstract syntax of the) datatype definition can be acquired using the reification facilities in Template Haskell.
- **Structure types**
  To avoid having to manipulate datatype elements directly, generic functions are defined over a structure type, instead of over the datatype itself. The

structure type reveals the top-level structure of the datatype and allows us to manipulate datatype elements in a uniform way. Both PolyP and Generic Haskell use a binary sum type to model multiple constructors and a binary product to model the arguments to the constructors. In this paper we use `Either` as the sum type and `(,)` as the product type.

When applied to the code for a datatype a generic function produces a function specialized for the structure type of that datatype. This means that we have to know how to translate between type codes and structure types. For instance, the type code `Par` in PolyP is translated to the structure type with the same name. Note that while there is a unique structure type for each type code, it is possible for several datatypes to have the same code (and thus the same structure).

– **Generic function definitions**

A generic function is implemented as a function from a type code to (abstract syntax for) the specialized version of the function. It might also be necessary to represent the type of a generic function in some way as we will see when implementing Generic Haskell in Section 5.

– **From structure types to datatypes**

The generic functions defined as described above produce functions specialized for the structure types. What we are interested in, on the other hand, are specializations for the actual datatypes. As described in Section 2, PolyP and Generic Haskell take two different approaches to constructing these specializations. In PolyP it is the responsibility of the user whereas in Generic Haskell, it is done by the compiler. In any case we need to be able to convert between an element of a datatype and an element of the corresponding structure type. How difficult the conversion functions are to generate depends on the complexity of the structure types. Both PolyP and Generic Haskell have quite simple structure types, so the only information we need to generate the conversion functions is the names and arities of the datatype constructors.

In the approach taken by PolyP, the conversion functions (`inn` and `out`) are all the compiler needs to define. The programmer of a generic function will then use these to lift her function from the structure type level to the datatype level. Implementing the Generic Haskell approach on the other hand requires some more machinery. For each generic function, the compiler must convert the specialization for a structure type into a function that operates on the corresponding datatype. In Section 5 we will see how to do this for Generic Haskell.

– **Instantiation**

Both the PolyP and Generic Haskell compilers do selective specialization, that is, generic functions are only specialized to the datatypes on which they are actually used in the program. This requires traversing the entire program to look for uses of generic functions. When embedding generic programming in Template Haskell we do not want to analyze the entire program to find out which specializations to construct. What we can do instead is to inline the body of the specialized generic function every time it is used. This makes the use of the generic functions easy, but special care has to be taken to avoid that recursive generic functions give rise to infinite specializations. This is the approach we use when embedding PolyP in Template Haskell (Section 4). Another option is to require the user to decide which functions to specialize on which datatypes. This makes it harder on the user, but a little easier for the implementor of the generic programming language. Since our focus is on fast prototyping of generic languages, this is the method we choose when implementing Generic Haskell.

4 PolyP in Template Haskell

Following the guidelines described in Section 3 we can start to implement our first generic programming language, PolyP.

4.1 Datatype Codes

The first thing to do is to decide on a datatype encoding. In PolyP, generic functions are defined by pattern matching on a pattern functor, so to get a faithful implementation of PolyP we should choose the type code to model these pattern functors.

```
data Code = Code :+: Code | Code :*: Code | Empty
          | Par | Rec | Regular :@: Code | Const Type
```

This coding corresponds perfectly to the definition of the pattern functors in Figure 1, we just have to decide what `Type` and `Regular` mean. The Template Haskell libraries define the abstract syntax for Haskell types in a datatype called `Type` so this is a natural choice to model types. The model of a regular datatype should contain a code for the corresponding pattern functor, but it also needs to contain the constructor names of the regular datatype. This is because we need to generate the `inn` and `out` functions for a regular datatype. Consequently we choose the following representation of a regular datatype:

```
type Regular = ([ConName], Code)
```

To make it easy to get hold of the code for a datatype, we want to define a function that converts from the (abstract syntax of a) datatype definition to `Regular`. A problem with this is that one regular datatype might depend on another regular datatype, in which case we have to look at the definition of the second datatype as well. So instead of just taking the definition of the datatype in question our conversion function takes a list of all definitions that might be needed together with the name of the type to be coded.

```
regular :: [Q Dec] -> TypeName -> Regular
```

Note that if `regular` fails, which it will if a required datatype definition is missing or the datatype is not regular, we will get a compile time error, since the function is executed by the Template Haskell system at compile time.

We can use the function `regular` to get the code for the `Rose` datatype defined in Section 1.2 as follows:

```
roseD = regular [reifyDecl Rose, reifyDecl []] "Rose"
```

4.2 Structure Types

The structure type of a regular datatype is a pattern functor. See Section 1.2 and Figure 1 in particular for their definitions. The mapping between the datatype codes (`Code`) and the pattern functors is the obvious one (a type code maps to the pattern functor with the same name).

4.3 Generic Function Definitions

Generic functions over pattern functors are implemented as functions from type codes to (abstract) Haskell code. For example, the function `fmap2` from Figure 2 in Section 1.2 is implemented as shown in Figure 5. The two definitions are strikingly similar, but there are a few important differences, the most obvious one being the splices and quasi-quote brackets introduced in the Template Haskell definition. Another difference is in the type signature. PolyP has its own type system capable of expressing the types of generic functions, but in Template Haskell everything inside quasi-quotes has type `Q Exp`, and thus the type of `fmap2` is lost. The third difference is that in Template Haskell we have to pass the type codes explicitly.

```
fmap2 :: Code -> Q Exp
fmap2 f =
  [| \p r -> $(
        case f of
          g :+: h -> [| $(fmap2 g) p r -+- $(fmap2 h) p r |]
          g :*: h -> [| $(fmap2 g) p r -*- $(fmap2 h) p r |]
          Empty   -> [| const () |]
          Par     -> [| p |]
          Rec     -> [| r |]
          d :@: g -> [| $(pmap d) ($(fmap2 g) p r) |]
          Const t -> [| id |]
      )
  |]
```

Fig. 5. fmap2 in Template Haskell

The `(:@:)`-case in the definition of `fmap2` uses the datatype level function `pmap` to map over the regular datatype `d`. The definition of `pmap` is described in Section 4.4.

4.4 From Structure Types to Datatypes

The generic functions described in Section 4.3 are defined over pattern functors, whereas the functions we are (really) interested in operate on regular datatypes. Somehow we must bridge this gap. In PolyP these datatype level functions are defined in terms of the pattern functor functions and the functions `inn` and `out`, that fold and unfold the top-level structure of a datatype. In PolyP, `inn` and `out` are primitive but in our setting they can be treated just like any generic function, that is, they take a code for a datatype and produce Haskell code. However, the code for a pattern functor is not sufficient, we also need to know the names of the constructors of the datatype in order to construct and deconstruct values of the datatype. This gives us the following types for `inn` and `out`.

```
inn, out :: Regular -> Q Exp
```

To see what code has to be generated we can look at the definition of `inn` and `out` for lists:

```
out_List :: [a] -> Either () (a, [a])
out_List = \xs -> case xs of
    []   -> Left ()
    x:xs -> Right (x,xs)

inn_List :: Either () (a, [a]) -> [a]
inn_List = \xs -> case xs of
    Left ()      -> []
    Right (x,xs) -> x:xs
```

Basically we have to generate a case expression with one branch for each constructor. In the case of `out` we match on the constructor and construct a value of the pattern functor whereas `inn` matches on the pattern functor and creates a value of the datatype. Note that the arguments to the constructors are left untouched, in particular the tail of the list is not unfolded.

With `inn` and `out` at our disposal we define the generic map function over a regular datatype, `pmap`. The definition is shown in Figure 6 together with the same definition in PolyP. In PolyP, `pmap` is a recursive function and we might be tempted to define it recursively in Template Haskell as well. This is not what we want, since it would make the generated code infinite, instead we want to *generate* a recursive function which we can do using a let binding.

4.5 Instantiation

The generic functions defined in this style are very easy to use. To map a function `f` over a rose tree `tree` we simply write

```
$(pmap roseD) f tree
```

where `roseD` is the representation of the rose tree datatype defined in Section 4.1.

```
pmap :: Regular d => (a -> b) -> d a -> d b
pmap f = inn . fmap2 f (pmap f) . out

pmap :: Regular -> Q Exp
pmap d = [| let pmap_d f = $(inn d)
                         . $(fmap2 $ functorOf d) f (pmap_d f)
                         . $(out d)
            in pmap_d
         |]
```

Fig. 6. The `pmap` function in PolyP and Template Haskell

5 Generic Haskell in Template Haskell

In Section 4 we outlined an embedding of PolyP into Template Haskell. In this section we do the same for the core part of Generic Haskell. Features of Generic Haskell that we do not consider include constructor cases, type indexed types and generic abstraction. Type indexed types and generic abstraction should be possible to add without much difficulty; constructor cases might require some work, though.

5.1 Datatype Codes

To be consistent with how generic functions are defined in Generic Haskell we choose the following datatype for type codes:

```
data Code = Sum | Prod | Unit
          | Con ConDescr | Label LabelDescr
          | Fun | TypeCon TypeName
          | App Code Code | Lam VarName Code | Var VarName
```

The first seven constructors should be familiar to users of Generic Haskell, although you do not see the `TypeCon` constructor when matching on a specific datatype in Generic Haskell. The last three constructors `App`, `Lam` and `Var` you never see in Generic Haskell. The reason why they are not visible is that the interpretation of these type codes is hard-wired into Generic Haskell and cannot be changed by the programmer. By making them explicit we get the opportunity to experiment with this default interpretation.

The types `ConDescr` and `LabelDescr` describe the properties of constructors and labels. In our implementation this is just the name, but it could also include information such as fixity and strictness.

If we define the infix application of `App` to be left associative, we can write the type code for the list datatype as follows:

```
listCode = Lam "a" $
  Sum `App` (Con "[]" `App` Unit)
      `App` (Con ":"
              `App` (Prod
                      `App` Var "a"
                      `App` (TypeCon "[]" `App` Var "a")
                    )
            )
```

This is not something we want to write by hand for every new datatype, even though it can be made much nicer by some suitable helper functions. Instead we define a function that produces a type code given the abstract syntax of a datatype declaration.

```
typeCode :: Q Dec -> Code
```

Thus, to get the above code for the list datatype we just write

```
typeCode (reifyDecl [])
```

5.2 Structure Types

The structure type for a datatype is designed to model the structure of that datatype in a uniform way. Similarly to PolyP, Generic Haskell uses binary sums and products to model datatype structures. We diverge from Generic Haskell in this implementation in that we use the Haskell prelude types `Either` and `(,)` for sums and products instead of defining our own. Another difference between our Template Haskell implementation and standard Generic Haskell is that constructors and labels are not modeled in the structure type. Compare, for example, the structure types of the list datatype in standard Generic Haskell (first) to our implementation (second):

```
type ListS a = Sum (Con Unit) (Con (Prod a [a])) -- std GH
type ListS a = Either    ()              (a,[a])  -- prototype
```

Since we are not implementing all features of Generic Haskell, we can allow ourselves this simplification.

5.3 Generic Function Definitions

The type of a generic function in Generic Haskell depends on the kind of the datatype the function is applied to. At a glance it would seem like we could ignore the types of a generic function, since Template Haskell does not have any support for typing anyway. It turns out, however, that we need the type when generating the datatype level functions. There are no type level lambdas in the abstract syntax for types in Template Haskell and there is no datatype for kinds. We need both these things when defining kind indexed types[2], so we define a

[2] type level lambdas are not strictly needed, but they make things much easier

```
data Kind = Star | FunK Kind Kind
data Type = ForallT [VarName] Context Type
          | VarT VarName | ConT ConName
          | TupleT Int | ListT | ArrowT
          | AppT Type Type
          | LamT [VarName] Type

a --> b = ArrowT `AppT` a `AppT` b
```

Fig. 7. Datatypes for kinds and types

datatype for kinds and a new datatype for types, shown in Figure 7. The kind indexed type `Map` from Section 1.3 can be defined as

```
_Map Star = LamT ["s","t"] $ VarT "s" --> VarT "t"
_Map (FunK k l) = LamT ["s","t"] $ ForallT ["a","b"] $
                    _Map k `AppT` a `AppT` b -->
                    _Map l `AppT` AppT s a `AppT` AppT t b
  where [s,t,a,b] = map VarT ["s","t","a","b"]
```

This is much clumsier than the Generic Haskell syntax, but we can make things a lot easier by observing that all type indexed types follow the same pattern. The only thing we need to know is the number of generic and non-generic arguments and the type for kind $\star$. With this information we define the function `kindIndexedType`:

```
kindIndexedType :: Int          -- # generic arguments
                -> Int          -- # non-generic arguments
                -> Type         -- type for kind *
                -> Kind -> Type
```

Using this function we define the type `Map` as

```
_Map = kindIndexedType 2 0 $ LamT ["s","t"]
                           $ VarT "s" --> VarT "t"
```

Now, the type of a generic function depends on the kind of the datatype it is applied to as well as the datatype itself. So we define a generic type to be a function from a kind and a type to a type.

```
type GenericType = Kind -> Type -> Type
```

The type of the generic map function from Section 1.3 can be defined as

```
gmapType :: GenericType
gmapType k t = _Map k `AppT` t `AppT` t
```

A generic function translates a type code to abstract Haskell syntax — we capture this in the type synonym `GenericFun`:

```
type GenericFun = Code -> Q Exp
```

To deal with types containing variables and abstraction, we also need an environment in which to store the translation of variables.

```
type GEnv        = [(VarName, Q Exp)]
type GenericFun' = GEnv -> GenericFun
```

With these types at our disposal we define the default translation, that will be the same for most generic functions. The function `defaultTrans`, defined in Figure 8, takes the name of the generic function that is being constructed and a `GenericFun'` that handles the non-standard translation and produces a `GenericFun'`. The idea is that a generic function should call `defaultTrans` on all type codes that it does not handle (see the generic map function in Figure 9 for an example).

```
defaultTrans :: VarName -> GenericFun' -> GenericFun'
defaultTrans name gfun env t = case t of
    Con _     -> [| id |]
    Label _   -> [| id |]
    TypeCon c -> varE $ gName name c
    App s t   -> [| $(gfun env s) $(gfun env t) |]
    Lam x t   -> [| \gx -> $(gfun ((x,[|gx|]):env) t) |]
    Var x     -> fromJust $ lookup x env

varE :: String -> Q Exp
```

Fig. 8. Default generic translations

The default translation for constructors and labels is the identity function. Since the structure type corresponding to `Con` and `Label` is the type level identity function of kind $\star \to \star$, a generic function applied to `Con` or `Label` expects a generic function for a type of kind $\star$ and should return a generic function for the same type.

The default translation of a named type is to call the specialized version of the generic function for that type. The function `gName` takes the name of a generic function and the name of a type and returns the name of the specialization of the generic function to that type. In our implementation it is the responsibility of the user to make sure that this specialization exists.

In Generic Haskell, the first three cases in `defaultTrans` can be changed by the programmer, for instance using the name of a constructor when pretty printing or defining special cases for particular types. The last three cases, on the other hand, are hidden from Generic Haskell users. A type level application is always translated into a value application, when encountering a type abstraction a generic function takes the translation of the abstracted variable as an argument, stores it in the environment and calls itself on the body of the abstraction. The translation of a type variable is looked up in the environment.

Provided that it does not need to change the default actions a generic function only has to provide actions for `Sum`, `Prod` and `Unit`. The definition of the generic map function from Section 1.3 is shown in Figure 9. For `Sum` and `Prod` the generic map function returns the map functions for `Either` and `(,)`, and mapping over the unit type is just the identity functions. For all other type codes we call the `defaultTrans` function to perform the default actions.

```
gmap :: GenericFun
gmap t = gmap' [] t
  where
    gmap' env t = case t of
      Sum  -> [| (-+-) |]
      Prod -> [| (-*-) |]
      Unit -> [| id |]
      t    -> defaultTrans "gmap" gmap' env t
```

Fig. 9. Generic map

5.4 From Structure Types to Datatypes

The generic functions defined in the style described in the previous section generate specializations for structure types, so from these structure type functions we have to construct functions for the corresponding datatypes. In our implementation of PolyP (Section 4) this was the responsibility of the programmer of the generic function. The reason for this was that in PolyP, the conversion could be done in several different ways, yielding different datatype functions. In Generic Haskell, on the other hand, we have a unique datatype function in mind for every structure type function. For instance, look at the type of the function generated by applying `gmap` to the code for the list datatype:

```
gmap_ListS :: (a -> b) -> Either () (a, [a])
                       -> Either () (b, [b])
gmap_ListS = $(gmap listS)
```

From this function we want to generate the map function for `List` with type

```
gmap_List :: (a -> b) -> [a] -> [b]
```

To be able to do this we first have to be able to convert between the `List` datatype and its structure type. For this purpose we define a function `structEP` that given the names and arities of a datatype's constructors generates the conversion functions between the datatype and its structure type.

```
structEP :: TypeName -> [(ConName, Int)] -> Q Dec
```

The `structEP` function generates a declaration of the conversion functions, so for the list datatype it would generate something like the following:

```
listEP :: EP [a] (Either () (a,[a]))
listEP = EP out inn
  where
    out []              = Left ()
    out (x:xs)          = Right (x,xs)
    inn (Left ())       = []
    inn (Right (x,xs)) = x:xs
```

The `EP` type, shown in Figure 10, models an embedding projection pair.

```
data EP a b = EP { from :: a -> b, to :: b -> a }

idEP :: EP a a
idEP = EP id id

funEP :: EP a a' -> EP b b' -> EP (a -> b) (a' -> b')
funEP epA epB = EP (\f -> from epB . f . to   epA)
                   (\g -> to   epB . g . from epA)
```

Fig. 10. Embedding projection pairs

Using `listEP` we define the map function for the list datatype as

```
gmap_List :: (a -> b) -> [a] -> [b]
gmap_List f = to (funEP listEP listEP) (gmap_ListS f)
```

The embedding projection pair is generated directly from the type of the generic function. In this case an embedding projection pair of type

```
EP ([a] -> [b]) (Either () (a,[a]) -> Either () (b,[b]))
```

should be generated. Embedding projection pairs between function types can be constructed with `funEP`, and `listEP` can convert between a list and an element of the list structure type. We define the function `typeEP` to generate the appropriate embedding projection pair.

```
typeEP :: Q Exp -> Kind -> GenericType -> Q Exp
```

The first argument to `typeEP` is the embedding projection pair converting between the datatype and its structure type, the second argument is the kind of the datatype and the third argument is the type of the generic function. So to get the embedding projection pair used in `gmap_List` we write

```
typeEP [| listEP |] (KFun Star Star) gmapType
```

5.5 Instantiation

The focus of this article is on fast prototyping of generic programming languages; this means that we do not make great efforts to facilitate the use of the generic functions. In particular what we do not do is figuring out which specializations to generate. Instead we provide a function `instantiate` that generates the definition of the specialization of a generic function to a particular datatype, as well as a function `structure`, that generates the embedding projection pair definition converting between a datatype and its structure type using the function `structEP` from Section 5.4.

```
type Generic  = (VarName, GenericType, GenericFun)
type Datatype = (TypeName, Kind, Code)

instantiate :: Generic -> Datatype -> Q [Dec]
structure   :: Datatype -> Q [Dec]
```

Using these functions a map function for rose trees can be generated by

```
data Rose a = Fork a [Rose a]
listD = ("[]", KFun Star Star, typeCode (reifyDecl []))
roseD = ("[]", KFun Star Star, typeCode (reifyDecl Rose))
gmapG = ("gmap", gmapType, gmap)
$(structure listD)
$(structure roseD)
$(instantiate gmapG listD)
$(instantiate gmapG roseD)
```

Since the rose trees contain lists we have to create specializations for the list datatype as well. The code generated for `gmap` specialized to rose trees will look something like the following (after some formatting and alpha renaming). Note that `gmap_RoseS` uses both `gmap_List` and `gmap_Rose`.

```
gmap_RoseS :: (a -> b) -> (a, [Rose a]) -> (b, [Rose b])
gmap_RoseS = \f -> f -*- gmap_List (gmap_Rose f)

gmap_Rose :: (a -> b) -> Rose a -> Rose b
gmap_Rose f = to (funEP roseEP roseEP) (gmap_RoseS f)
```

6 Conclusions and Future Work

Efforts to explore the design space of generic programming have been hampered by the fact that implementing a generic programming language is a daunting task. In this paper we have shown that this does not have to be the case. We have presented two prototype implementations of generic programming approximating PolyP and Generic Haskell. Thanks to the Template Haskell machinery, these prototypes could be implemented in a short period of time (each implementation consists of a few hundred lines of Haskell code). Comparing these two implementations we obtain a better understanding of the design space when it comes to implementations of generic programming languages.

There are a few different areas one might want to focus future work on:

- The idea of fast prototyping is to make it possible to experiment with different ideas in an easy way. So far, most of our work has been concentrated on how to write the prototypes and not so much on experimenting.
- One of the biggest problems with current generic programming systems is efficiency. The conversions between datatypes and structure types takes a lot of time and it would be a big win if one could remove this extra cost. We have started working on a simplifier for Haskell expressions that can do this.
- It would be interesting to see how other generic programming styles fit into this framework. In particular one could look at the `Data.Generics` libraries in GHC [13] and also at the generic traversals of adaptive OOP [14].
- The design of a generic programming language includes the design of a type system. In this paper we have ignored the issue of typing, leaving it up to the Haskell compiler to find type errors in the specialized code. Is there an easy way to build prototype type systems for our implementations?

References

1. A. Alimarine and R. Plasmeijer. A generic programming extension for Clean. In T. Arts and M. Mohnen, editors, *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001*, volume 2312 of *LNCS*, pages 168–185. Springer-Verlag, 2001.
2. D. Clarke and A. Löh. Generic haskell, specifically. In J. Gibbons and J. Jeuring, editors, *Proceedings of the IFIP TC2 Working Conference on Generic Programming*, pages 21–48. Kluwer, 2003.
3. R. Cockett and T. Fukushima. About Charity. Yellow Series Report No. 92/480/18, Dep. of Computer Science, Univ. of Calgary, 1992.
4. R. Hinze. Polytypic values possess polykinded types. In *Mathematics of Program Construction*, volume 1837 of *LNCS*, pages 2–27. Springer-Verlag, 2000.
5. R. Hinze and J. Jeuring. Generic Haskell: Applications. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 57–97. Springer-Verlag, 2003.
6. R. Hinze and J. Jeuring. Generic Haskell: Practice and theory. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 1–56. Springer-Verlag, 2003.
7. R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science, 2001.
8. G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.
9. P. Jansson. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden, May 2000.
10. P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *POPL'97*, pages 470–482. ACM Press, 1997.
11. P. Jansson and J. Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

12. C. Jay and P. Steckler. The functional imperative: shape! In C. Hankin, editor, *Programming languages and systems: 7th European Symposium on Programming, ESOP'98*, volume 1381 of *LNCS*, pages 139–53. Springer-Verlag, 1998.
13. R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *SIGPLAN Not.*, 38(3):26–37, 2003.
14. K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.
15. U. Norell and P. Jansson. Polytypic programming in Haskell. In *Implementation of Functional Languages*, LNCS, 2004. In press for LNCS. Presented at IFL'03.
16. T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the Haskell workshop*, pages 1–16. ACM Press, 2002.