Functional Generic Programming and Type Theory

Ulf Norell, `ulfn@cs.chalmers.se`

October 24, 2002

Abstract. Functional generic programming is an area of research concerning programs parameterized by types. Such parameterization is a powerful method of abstraction that allows the programmer to define and reuse common patterns of computation that work over many different datatypes.

This thesis investigates two ways of implementing generic programming; by simulating it in a smaller language without generic constructs and by embedding it in a more powerful language. For the first approach we explore the possibility of simulating PolyP and Generic Haskell in Haskell and for the second approach we embed Generic Haskell into the dependently typed Alfa/AGDA system.

Contents

Chapter 1

Introduction

Functional programming draws great power from the ability to define polymorphic, higher order functions that can capture the structure of an algorithm while abstracting away from the details. A polymorphic function is parameterized over one or more types and thus abstracting away from the specifics of these types. The same is true for a functional generic function, but while all instances of a polymorphic function share the same definition, the instances of a generic function definition also depend on a type.

By parameterizing the function definition by a type one can capture common patterns of computation over different datatypes. Examples of functions that can be defined generically include the map function that maps a function over a datatype but also more complex algorithms like unification and term rewriting.

Even if an algorithm will only be applied to a single datatype it may still be a good idea to implement it as a generic function. First of all, since a generic function abstracts away from the details of the datatype we cannot make any datatype specific mistakes in the definition and secondly, if the datatype changes, there is no need to change the generic function.

In this thesis we explore two ways of implementing functional generic programming; first by simulating it in a less powerful language without generic constructs (Haskell) and second by embedding it in a bigger language with dependent types (Alfa).

1.1 Related work

We use two systems for generic programming in this thesis; PolyP [Jan00] and Generic Haskell [HJ]. PolyP adds a generic construct to a subset of the functional language Haskell [PeHeA$^+$99] and provides a compiler that compiles PolyP programs into Haskell. The generic functions in PolyP are restricted to regular datatypes with one parameter. Generic Haskell is the successor of PolyP and extends full Haskell 98 with the ability to write generic functions over arbitrary datatypes. The Generic Haskell compiler generates standard Haskell code from the generic program.

The system for dependently typed programming we use is the Alfa/AGDA system. Alfa [HR00] is the proof editor and AGDA [CC99] is the underlying language. For simplicity we henceforth refer to the entire framework as Alfa.

Pfeifer and Rueß [PR99] use type theory to represent generic programs introducing a library of operators with which they construct the generic map function for datatypes of first order kinds. Their approach however is not easily extended to other generic functions or datatypes of higher ordered kinds.

Hinze and Peyton Jones [HP01] describes an simple extension to the Haskell class system that allows the programmer to specify how to derive an instance of a specific class for datatypes of kind $\star$. Their work relates to chapters 3 and 5 where we try to use the Haskell class system as it is to implement generic programming.

In his master's thesis [Sin01] Sinot describes an embedding of PolyP in the dependently typed language Cayenne. In chapter 7 we embed Generic Haskell into Alfa.

Altenkirch and McBride [AM02] show how to encode higher kinded generic programming in OLEG, a dependently typed programming system, by introducing a universe for datatype representations. Their results are similar to those in chapter 7.

Benke is currently working on an implementation of generic programming in Alfa that allows generic functions over dependent datatypes.

1.2 Overview

This thesis contains a discussion on how to use the Haskell class system to implement generic programming and an embedding of generic programming in type theory.

Chapter 1 contains an overview of this thesis and of related work and an introduction to some basic concepts.

Chapter 2 introduces the PolyP system and chapter 3 introduces the class translation, that translates a generic program into a Haskell program were we use the class system to implement genericity, and describes a modified version of the PolyP compiler implementing this translation.

Chapter 4 gives a brief introduction to Generic Haskell and chapter 5 describes an attempt to formulate the class translation for Generic Haskell.

Chapter 6 introduces type theory and the Alfa [HR00] framework and chapter 7 describes an embedding of Generic Haskell programs in Alfa.

1.3 Basic concepts

This section gives a brief introduction to some concepts that are being used in the rest of the thesis. Each subsection introduces a Haskell type and possibly some functions operating over it.

1.3.1 Function types

The type of functions from the type a to the type b is denoted by $a \to b$. Functions can be defined by lambda abstraction, so $\lambda x.\, x+1$ is an element of the type $\mathsf{Int} \to \mathsf{Int}$. Haskell allows higher order functions, so we can write functions that operates on functions. For instance we can define function composition as follows

$$\begin{array}{lcl} (\circ) & :: & (a \to b) \to (c \to a) \to c \to b \\ (f \circ g)\ x & = & f\ (g\ x) \end{array}$$

Note that the function arrow associates to the right.

1.3.2 Binary products

The binary product of types a and b is written $(a,\, b)$. The same notation is used for elements of the product type, so $(4,\, 2)$ is an element of the type $(\mathsf{Int},\, \mathsf{Int})$. To extract information from a pair there are two projection functions

$$\begin{array}{lcl} \mathit{fst} & :: & (a,\, b) \to a \\ \mathit{fst}\ (x,\, y) & = & x \\ & & \\ \mathit{snd} & :: & (a,\, b) \to b \\ \mathit{snd}\ (x,\, y) & = & y \end{array}$$

We also define a map function $(-\!\!*\!\!-)$ that takes two functions and applies them to the components of a pair

$$\begin{array}{lcl} (-\!\!*\!\!-) & :: & (a \to c) \to (b \to d) \to (a,\, b) \to (c,\, d) \\ (f -\!\!*\!\!- g)\ (x,\, y) & = & (f\ x,\, g\ y) \end{array}$$

1.3.3 Nullary product

The nullary product type () (pronounced unit) contains the one element ().

1.3.4 Disjoint sums

In Haskell the disjoint sum of the types a and b is $\mathsf{Either}\ a\ b$, where

$$\begin{array}{lcl} \mathbf{data}\ \mathsf{Either}\ a\ b & = & \mathsf{Left}\ a \\ & \mid & \mathsf{Right}\ b \end{array}$$

An element of type $\mathsf{Either}\ a\ b$ is either of the form $\mathsf{Left}\ x$ where $x :: a$ or $\mathsf{Right}\ y$ where $y :: b$. We define a map function $(-\!\!+\!\!-)$ similar to $(-\!\!*\!\!-)$ that takes two functions and apply the appropriate one to the element of a disjoint sum.

$$\begin{array}{lcl} (-\!\!+\!\!-) & :: & (a \to c) \to (b \to d) \to \mathsf{Either}\ a\ b \to \mathsf{Either}\ c\ d \\ (f -\!\!+\!\!- g)\ (\mathsf{Left}\ x) & = & \mathsf{Left}\ (f\ x) \\ (f -\!\!+\!\!- g)\ (\mathsf{Right}\ y) & = & \mathsf{Right}\ (g\ y) \end{array}$$

1.3.5 Embedding-projection pairs

An embedding-projection pair embedding a in b is a pair of functions *from* and *to* such that $to \circ from = id$. In Haskell we can define the type of embedding-projection pairs as follows

$$\begin{array}{lclll} \mathbf{data}\ \mathsf{EP}\ a\ b & = & \mathsf{EP} & \{ & from :: a \to b \\ & & & , & to :: b \to a \\ & & & \} & \end{array}$$

When we defined the sum and product types we also defined the map functions $(-\!\!+\!\!-)$ and $(-\!\!*\!\!-)$. We can generalize these functions to work over embedding-projection pairs instead of functions. This generalization also allows us to define such a function for the function type.

$$\begin{array}{lcl} (\xrightarrow{ep}) & :: & \mathsf{EP}\ a\ c \to \mathsf{EP}\ b\ d \to \mathsf{EP}\ (a \to b)\ (c \to d) \\ ep_1 \xrightarrow{ep} ep_2 & = & \mathsf{EP}\ (\lambda f.\, from\ ep_2 \circ f \circ to\ ep_1) \\ & & \quad (\lambda g.\, to\ ep_2 \circ g \circ from\ ep_1) \\ & & \\ ep_{\mathsf{Either}} & :: & \mathsf{EP}\ a\ c \to \mathsf{EP}\ b\ d \to \mathsf{EP}\ (\mathsf{Either}\ a\ b)\ (\mathsf{Either}\ c\ d) \\ ep_{\mathsf{Either}}\ ep_1\ ep_2 & = & \mathsf{EP}\ (from\ ep_1 -\!\!+\!\!- from\ ep_2) \\ & & \quad (to\ ep_1 -\!\!+\!\!- to\ ep_2) \\ & & \\ ep_{(,)} & :: & \mathsf{EP}\ a\ c \to \mathsf{EP}\ b\ d \to \mathsf{EP}\ (a, b)\ (c, d) \\ ep_{(,)}\ ep_1\ ep_2 & = & \mathsf{EP}\ (from\ ep_1 -\!\!*\!\!- from\ ep_2) \\ & & \quad (to\ ep_1 -\!\!*\!\!- to\ ep_2) \end{array}$$

We can, in a similar way, lift the *fmap* function to embedding-projection pairs rendering the function ep_{Functor}:

$$\begin{array}{lcl} ep_{\mathsf{Functor}} & :: & \mathsf{Functor}\ f \Rightarrow \mathsf{EP}\ a\ b \to \mathsf{EP}\ (f\ a)\ (f\ b) \\ ep_{\mathsf{Functor}}\ ep & = & \mathsf{EP}\ (fmap\ (from\ ep))\ (fmap\ (to\ ep)) \end{array}$$

We also define the identity embedding-projection pair ep_{id} as

$$\begin{array}{lcl} ep_{id} & :: & \mathsf{EP}\ a\ a \\ ep_{id} & = & \mathsf{EP}\ id\ id \end{array}$$

Chapter 2

PolyP

2.1 Introduction

PolyP is an extension to (a subset of) Haskell that allows definitions of polytypic functions over regular, unary datatypes. PolyP was developed as a part of Jansson's PhD thesis [Jan00]. In this chapter we give a short overview of PolyP, section 2.2 introduces the notion of viewing datatypes as fixpoints, section 2.3 describes how to write polytypic functions and section 2.4 looks briefly at the PolyP compiler.

2.2 Datatypes as fixed points

As mentioned earlier PolyP allows the definition of polytypic functions over regular datatypes of kind $\star \to \star$. A datatype is regular if it is not mutually recursive, does not contain any function spaces and the arguments to the type constructor are the same in the left-hand side and the right-hand side of the definition.

We can view a regular datatype $D\ a$ as the least fixed point of a binary functor.

$$\begin{array}{lcl}
\mathbf{type}\ (g + h)\ p\ r & = & \mathsf{Either}\ (g\ p\ r)\ (h\ p\ r) \\
\mathbf{type}\ (g * h)\ p\ r & = & (g\ p\ r,\ h\ p\ r) \\
\mathbf{type}\ \mathsf{Empty}\ p\ r & = & () \\
\mathbf{type}\ \mathsf{Par}\ p\ r & = & p \\
\mathbf{type}\ \mathsf{Rec}\ p\ r & = & r \\
\mathbf{type}\ (d @ g)\ p\ r & = & d\ (g\ p\ r) \\
\mathbf{type}\ \mathsf{Const}\ t\ p\ r & = & t
\end{array}$$

Figure 2.1: Pattern functors

$$\begin{array}{lll} inn & :: & \mathsf{Regular}\ d \Rightarrow \Phi_d\ a\ (d\ a) \to d\ a \\ out & :: & \mathsf{Regular}\ d \Rightarrow d\ a \to \Phi_d\ a\ (d\ a) \end{array}$$

Figure 2.2: The *inn* and *out* functions

For instance the datatype List a defined by

$$\textbf{data}\ \mathsf{List}\ a = \mathsf{Nil} \mid \mathsf{Cons}\ a\ (\mathsf{List}\ a)$$

can be viewed as the least fixed point of the datatype ListF a r defined by

$$\textbf{data}\ \mathsf{ListF}\ a\ r = \mathsf{NilF} \mid \mathsf{ConsF}\ a\ r$$

This functor is obtained simply by abstracting over the recursive call in the definition of List. Now since we only want ListF to describe the structure of List we can discard the constructor names and define ListF as a type synonym

$$\textbf{type}\ \mathsf{ListF}\ a\ r = \mathsf{Either}\ ()\ \ (a, r)$$

Here Left () corresponds to an empty list (NilF) and Right (x, xs) corresponds to the non-empty list ConsF x xs. The type ListF is called a *pattern functor* since it describes the recursion pattern of a datatype. We can write ListF in a point-free style, using the pattern functors defined in figure 2.1.

$$\textbf{type}\ \mathsf{ListF} = \mathsf{Empty} + \mathsf{Par} * \mathsf{Rec}$$

The pattern functor (@) is used to compose a regular datatype and a pattern functor, so can describe the structure of the datatype Rose as follows

$$\begin{array}{lll} \textbf{data}\ \mathsf{Rose}\ a & = & \mathsf{Fork}\ a\ (\mathsf{List}\ (\mathsf{Rose}\ a)) \\ \textbf{type}\ \mathsf{RoseF} & = & \mathsf{Par} * \mathsf{List@Rec} \end{array}$$

The last pattern functor Const captures constant types in datatype definition so can define

$$\begin{array}{lll} \textbf{data}\ \mathsf{Tree}\ a & = & \mathsf{Leaf}\ a \mid \mathsf{Branch}\ \mathsf{Int}\ (\mathsf{Tree}\ a)\ (\mathsf{Tree}\ a) \\ \textbf{type}\ \mathsf{TreeF} & = & \mathsf{Par} + \mathsf{Const}\ \mathsf{Int} * \mathsf{Rec} * \mathsf{Rec} \end{array}$$

for a tree with an explicit height field.

In general we write Φ_D for the pattern functor of the datatype D, so in this case $\Phi_{\mathsf{List}} = \mathsf{ListF}$. To convert between a datatype and its pattern functor we use the built-in functions *inn* and *out* shown in figure 2.2. These functions realize the isomorphism $\Phi_d\ a\ (d\ a) \cong d\ a$, that holds for every regular datatype.

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

Figure 2.3: The polytypic map function *fmap*2

In our list example we have

```
inn_List :: ListF a (List a) → List a
inn_List (Left ())          =  Nil
inn_List (Right (x, xs))    =  Cons x xs

out_List :: List a → ListF a (List a)
out_List Nil                =  Left ()
out_List (Cons x xs)        =  Right (x, xs)
```

2.3 Defining polytypic functions

In the previous section we introduced the pattern functor Φ_d of a regular datatype d. In this section we show how to write polytypic functions by induction of this pattern functor. To this end PolyP introduces a type case construct that allows us to write functions that pattern match over the structure of the pattern functor. Figure 2.3 shows the definition of a polytypic map function over pattern functors. We can then define a polytypic map over regular datatypes using *fmap*2 in the following way

```
pmap    ::  Regular d ⇒ (a → b) → d a → d b
pmap f  =   inn ∘ fmap2 f (pmap f) ∘ out
```

In summary, the **polytypic** construct allows us to write polytypic functions over pattern functors by induction over the structure of the pattern functor. We can then use these functions together with the functions *inn* and *out* to define functions that work on all regular datatypes.

2.4 Behind the scenes

The PolyP compiler takes a PolyP program a compiles it into Haskell 98 code. This is done by analyzing the program to determine at which datatypes poly-

$$
\begin{array}{lll}
\mathit{pmap}_{\mathsf{List}} & :: & (a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b \\
\mathit{pmap}_{\mathsf{List}}\ f & = & \mathit{inn}_{\mathsf{List}} \circ \mathit{fmap2}_{\mathsf{ListF}}\ f\ \mathit{pmap}_{\mathsf{List}} \circ \mathit{out}_{\mathsf{List}} \\
\\
\mathit{fmap2}_{\mathsf{ListF}} & :: & (a \to c) \to (b \to d) \to \mathsf{Either}\ ()\ \ (a,\, b) \to \mathsf{Either}\ ()\ \ (c,\, d) \\
\mathit{fmap2}_{\mathsf{ListF}} & = & \lambda p\ r \to \mathit{fmap2}_{\mathsf{Empty}}\ p\ r \mathbin{-\!\!+\!\!-} \mathit{fmap2}_{\mathsf{Par*Rec}}\ p\ r \\
\\
\mathit{fmap2}_{\mathsf{Empty}} & :: & (a \to c) \to (b \to d) \to () \to () \\
\mathit{fmap2}_{\mathsf{Empty}} & = & \lambda p\ r \to \mathit{id} \\
\\
\mathit{fmap2}_{\mathsf{Par*Rec}} & :: & (a \to c) \to (b \to d) \to (a,\, b) \to (c,\, d) \\
\mathit{fmap2}_{\mathsf{Par*Rec}} & = & \lambda p\ r \to \mathit{fmap2}_{\mathsf{Par}}\ p\ r \mathbin{-\!\!*\!\!-} \mathit{fmap2}_{\mathsf{Rec}}\ p\ r \\
\\
\mathit{fmap2}_{\mathsf{Par}} & :: & (a \to c) \to (b \to d) \to a \to c \\
\mathit{fmap2}_{\mathsf{Par}} & :: & \lambda p\ r \to p \\
\\
\mathit{fmap2}_{\mathsf{Rec}} & :: & (a \to c) \to (b \to d) \to b \to d \\
\mathit{fmap2}_{\mathsf{Rec}} & :: & \lambda p\ r \to r
\end{array}
$$

Figure 2.4: The function *pmap* specialized to lists

typic functions are used and then generating specialized instances of the polytypic functions for these datatypes. For instance if we only use the function *pmap* on the datatype List in our program, we would get the specialized functions shown in figure 2.4, together with the instances of *inn* and *out* for lists shown in the previous section.

Chapter 3

Class translation for PolyP

As described in section 2 a PolyP program is compiled into Haskell by specializing. Each polytypic function is specialized with respect to the types it is used at and calls to the polytypic function are replaced by calls to the corresponding specialized version. In this section we describe an alternative method of compiling PolyP programs, which we term *class translation*, using the Haskell class system with the extension of multiple parameter classes and functional dependencies.

The idea of the class translation is to create a class for each polytypically defined function, that is for every function defined using the **polytypic** keyword. The different branches of the type case is translated to instance declaration of the new class for the corresponding pattern functor (section 3.3). We also introduce the class FunctorOf (section 3.2) to tie together a datatype and its structure.

3.1 Motivation

From the PolyP programmer's view it doesn't really matter what the compiled code looks like as long as it is reasonably efficient, and since we do not really expect to get faster code using the class system, one could question the need for an alternative to specialization. Our motivation is that we want to use polytypic functions in regular Haskell programs and not only in PolyP programs. We also want to be able to do separate compilation of polytypic modules. The current method of specialization fails with both these tasks.

We can classify PolyP modules into three categories; modules that *define* new polytypic functions, modules that *use* polytypic functions and standard Haskell modules. Using a specializing compiler we have to compile at least the modules of the first and second category, and since the current implementation of the PolyP compiler cannot handle full Haskell we cannot use polytypic functions in *real* Haskell programs.

```
data SumF g h p r   =  InL (g p r)
                    |  InR (h p r)
data ProdF g h p r  =  g p r :*: h p r
data EmptyF p r     =  Empty
data ParF p r       =  Par {unParF :: p}
data RecF p r       =  Rec {unRecF :: r}
data CompF d g p r  =  Comp {unCompF :: d (g p r)}
data ConstF t p r   =  Const {unConstF :: t}

unSumF              :: SumF g h p r → Either (g p r) (h p r)
unSumF (InL x)      =  Left x
unSumF (InR y)      =  Right y

unProdF             :: ProdF g h p r → (g p r, h p r)
unProdF (x :*: y)   =  (x, y)

unEmptyF            :: EmptyF p r → ()
unEmptyF Empty      =  ()
```

Figure 3.1: Datatype versions of the pattern functors together with functions to convert to the type synonym versions

```
class FunctorOf f d | d → f where
  inn  ::  f a (d a) → d a
  out  ::  d a → f a (d a)
```

Figure 3.2: The FunctorOf class

Ideally we would only need to compile modules of the first category, thus making it possible to write a library of polytypic functions, compile it into Haskell and then use it just like any library of regular Haskell functions. The method described in this chapter comes quite close to this ideal, requiring only small (automatable) additions to the Haskell program using polytypic functions.

3.2 Representing datatypes

In PolyP the pattern functors are just type synonyms (see figure 2.1), but since we want to generate class instances for the pattern functors we have to use real datatypes. Our new functors are shown in figure 3.1.

We use the class FunctorOf to implement the isomorphism between a type and its top level structure. If we compare the *inn* and *out* functions in the FunctorOf class (figure 3.2) with the *inn* and *out* functions defined in PolyP (figure 2.2) we can see that the only difference is that we have given a name to the pattern functor Φ_d. The advantage of this similarity is that we can use these functions in ordinary Haskell code in the same way as they are used in PolyP which means that we do not have to change uses of *inn* and *out* when compiling the PolyP code. Of course, we do have to change the types as seen below. The gritty details of changing the types are described in section 3.4.

The FunctorOf class uses a functional dependency stating that for each datatype there is a unique pattern functor. This dependency is crucial to this method since without it we cannot type functions where the pattern functor does not occur in the type, as for example the polytypic map function

```
pmap :: Regular d ⇒ (a → b) → d a → d b
pmap f = inn ∘ fmap2 f (pmap f) ∘ out
```

When we compile this function into Haskell we get

```
pmap :: (FunctorOf f d, P_fmap2 f) ⇒ (a → b) → d a → d b
pmap f = inn ∘ fmap2 f (pmap f) ∘ out
```

Here P_fmap2 is the class generated from the polytypic function *fmap*2. Since the pattern functor f does not occur in the type body there would be no way of inferring its value without the functional dependency.

To be able to use our polytypic functions with a datatype we need an instance of FunctorOf for that particular type and its pattern functor. These instances can be generated automatically in a fairly straightforward way. First lets take a look at the generated instance for the list datatype.

```
instance FunctorOf (SumF EmptyF (ProdF ParF RecF)) [] where
  inn (InL Empty)             =  []
  inn (InR (Par a :*: Rec b)) =  a : b
  out []                      =  InL Empty
  out (a : b)                 =  InR (Par a :*: Rec b)
```

The function *out* pattern matches on the datatype value and returns the appropriate element of the pattern functor, while *inn* does the inverse.

This example covers all but two of the pattern functors, CompF and ConstF. The ConstF case can be handled in the same way as RecF and ParF, but CompF is a little trickier. Lets look at another example, the Rose datatype.

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

The pattern functor of Rose is

```
RoseF = ProdF ParF (CompF [] RecF)
```

An element of RoseF a (Rose a) is of the form Par x :*: Comp ys where x :: a and ys :: [RecF a (Rose a)]. The problem is that the Fork constructor expects a

$$\begin{array}{lcl}
unF & \in & PatternFunctor \to Code \\
unF(\mathsf{ParF}) & = & \text{“}unParF\text{”} \\
unF(\mathsf{RecF}) & = & \text{“}unRecF\text{”} \\
unF(\mathsf{ConstF}) & = & \text{“}unConstF\text{”} \\
unF(\mathsf{CompF}\ d\ g) & = & \text{“}pmap\ (\text{”} + unF(g) + \text{“}) \circ unCompF\text{”}
\end{array}$$

$$\begin{array}{lcl}
nuF & \in & PatternFunctor \to Code \\
nuF(\mathsf{ParF}) & = & \text{“}\mathsf{Par}\text{”} \\
nuF(\mathsf{RecF}) & = & \text{“}\mathsf{Rec}\text{”} \\
nuF(\mathsf{ConstF}) & = & \text{“}\mathsf{Const}\text{”} \\
nuF(\mathsf{CompF}\ d\ g) & = & \text{“}\mathsf{Comp} \circ pmap\ (\text{”} + nuF(g) + \text{“})\text{”}
\end{array}$$

Figure 3.3: Meta function *unF* and *nuF*

value of type [Rose a]. So we have to convert between values of type [Rose a] and [RecF a (Rose a)] or in the general case between d' $(\hat{g}\ a\ (d\ a))$ and d' $(g\ a\ (d\ a))$, where d' is the datatype in the composition ([] in our example), g is the pattern functor that d' is composed with (RecF) and d is the datatype containing the composition (Rose). We let $\hat{g}$ denote the pattern functor g interpreted as a type synonym. Since we know that d' is a regular datatype, we can use the polytypic map function *pmap* to do the conversion. Remember that with the class translation we can use polytypic functions directly in our Haskell code. We can define two (meta) functions *unF* and *nuF* that handles the conversion (see figure 3.3). These functions take the pattern functor g of a composition as an argument and return conversion functions between g and $\hat{g}$. Note that SumF and ProdF cannot occur inside a composition. Using these functions we get the following FunctorOf instance for Rose.

```
instance FunctorOf (ProdF ParF (CompF [] RecF)) Rose where
  inn (Par a :*: Comp b)  =  Fork a (pmap unRec b)
  out (Fork a b)          =  Par a :*: Comp (pmap Rec b)
```

After these examples we can look at the class translation algorithm in more detail. A regular datatype D can be written on the following form.

$$\begin{array}{llllll}
\textbf{data}\ D\ a & = & C_1 & (\widehat{g_{11}}\ a\ (D\ a)) & \dots & (\widehat{g_{1m_1}}\ a\ (D\ a)) \\
 & \vdots & & & & \\
 & | & C_n & (\widehat{g_{n1}}\ a\ (D\ a)) & \dots & (\widehat{g_{nm_n}}\ a\ (D\ a))
\end{array}$$

In which case the corresponding pattern functor (expressed in PolyP notation) is

$$\Phi_D = (g_{11} * \ldots * g_{1m_1}) + \ldots + (g_{n1} * \ldots * g_{nm_n})$$

Here C_i is a constructor of D and g_{ij} is a pattern functor not containing sums or products. A FunctorOf instance for D is shown in figure 3.4. The meta functions

```
instance FunctorOf $\Phi_D$ $D$ where
  $inn\ (lhs(g_{11}, a_1)$ :*: ... :*: $lhs(g_{1m_1}, a_{m_1})$          )
    $= C_1\ (rhs_{inn}(g_{11}, a_1)) \ldots (rhs_{inn}(g_{1m_1}, a_{m_1}))$
  $\vdots$
  $inn\ (lhs(g_{n1}, a_1)$ :*: ... :*: $lhs\ g_{nm_n}\ a_{m_n})$
    $= C_n\ (rhs_{inn}(g_{n1}, a_1)) \ldots (rhs_{inn}(g_{nm_n}, a_{m_n}))$

  $out\ (C_1\ a_1 \ldots a_{m_1})$
    $= rhs_{out}(g_{11}, a_1)$ :*: ... :*: $rhs_{out}(g_{1m_1}, a_{m_1})$
  $\vdots$
  $out\ (C_n\ a_1 \ldots a_{m_n})$
    $= rhs_{out}(g_{n1}, a_1)$ :*: ... :*: $rhs_{out}(g_{nm_n}, a_{m_n})$
```

Figure 3.4: The FunctorOf instance for a generic datatype

lhs, rhs_{inn} and rhs_{out} are defined in figure 3.5. The function lhs takes a pattern functor g and a variable name and returns a pattern, matching an element of $g\ a\ b$ for any a and b. The functions rhs_{inn} and rhs_{out} take a pattern functor g and a variable name a and generate the code in the right hand side of *inn* and *out* respectively, corresponding to the pattern functor g.

3.3 The polytypic construct

Remember from chapter 2 that polytypic functions are defined using the **polytypic** keyword. The definition of the polytypic function *fmap*2 from figure 2.3 is shown again in figure 3.6. Our compiler translates a polytypic definition into a class with one member (the polytypic function) and instance declarations for the pattern functors corresponding to the branches in the definition. A compiled version of the *fmap*2 function is shown in figure 3.7. Note that the type of the class member *fmap*2 is the same as the type of the polytypic function *fmap*2, which makes it possible to use the compiled version in the same way as the original function was used.

The type case in the polytypic definition is compiled into instance declarations for the pattern functors in the branches. In some cases there are constraints on the instances, for example SumF g h is an instance of P_fmap2 only if both g and h are instances of P_fmap2. These constraints are inferred from the actual definition of the instance, in this case the use of *fmap*2 on both sides of the sum. This inference is described below.

The only tricky part here is that the polytypic definition views the pattern functors as type synonyms (figure 2.1), while the data type interpretation (figure 3.1) has to be used in the instance declarations. This means that we cannot use the definitions from the PolyP code as they are, but we have to convert them to the appropriate type. To do this we define embedding-projection pairs between

$$\begin{array}{lcl}
lhs & \in & PatternFunctor \times Name \rightarrow Pattern \\
lhs(\mathsf{EmptyF}, a) & = & \text{“}\mathsf{Empty}\text{”} \\
lhs(\mathsf{ParF}, a) & = & \text{“}\mathsf{Par}\ a\text{”} \\
lhs(\mathsf{RecF}, a) & = & \text{“}\mathsf{Rec}\ a\text{”} \\
lhs(\mathsf{ConstF}, a) & = & \text{“}\mathsf{Const}\ a\text{”} \\
lhs(\mathsf{CompF}\ d\ g, a) & = & \text{“}\mathsf{Comp}\ a\text{”} \\
\\
rhs_{inn} & \in & PatternFunctor \times Name \rightarrow Code \\
rhs_{inn}(\mathsf{ParF}, a) & = & \text{“}a\text{”} \\
rhs_{inn}(\mathsf{RecF}, a) & = & \text{“}a\text{”} \\
rhs_{inn}(\mathsf{ConstF}, a) & = & \text{“}a\text{”} \\
rhs_{inn}(\mathsf{CompF}\ d\ g, a) & = & \text{“}pmap\ (\text{”} + unF(g) + \text{“})\ a\text{”} \\
\\
rhs_{out} & \in & PatternFunctor \times Name \rightarrow Code \\
rhs_{out}(\mathsf{EmptyF}, a) & = & \text{“}\mathsf{Empty}\text{”} \\
rhs_{out}(\mathsf{ParF}, a) & = & \text{“}\mathsf{Par}\ a\text{”} \\
rhs_{out}(\mathsf{RecF}, a) & = & \text{“}\mathsf{Rec}\ a\text{”} \\
rhs_{out}(\mathsf{ConstF}, a) & = & \text{“}\mathsf{Const}\ a\text{”} \\
rhs_{out}(\mathsf{CompF}\ d\ g, a) & = & \text{“}\mathsf{Comp}\ (pmap\ (\text{”} + nuF(g) + \text{“})\ a)\text{”}
\end{array}$$

Figure 3.5: Meta functions used in FunctorOf instance generation

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

Figure 3.6: The polytypic map function *fmap*2

class $\mathsf{P\_fmap2}\ f$ **where**
$\quad fmap2 \quad :: \quad (a \to c) \to (b \to d) \to (f\ a\ b \to f\ c\ d)$

instance $(\mathsf{P\_fmap2}\ g, \mathsf{P\_fmap2}\ h) \Rightarrow \mathsf{P\_fmap2}\ (\mathsf{SumF}\ g\ h)$ **where**
$\quad fmap2 \quad = \quad from\ (ep_{id} \xrightarrow{ep} ep_{id} \xrightarrow{ep} ep_{+} \xrightarrow{ep} ep_{+})$
$\qquad\qquad\qquad (\lambda\ p\ r \to fmap2\ p\ r \dashv\!\vdash fmap2\ p\ r)$

instance $(\mathsf{P\_fmap2}\ g, \mathsf{P\_fmap2}\ h) \Rightarrow \mathsf{P\_fmap2}\ (\mathsf{ProdF}\ g\ h)$ **where**
$\quad fmap2 \quad = \quad from\ (ep_{id} \xrightarrow{ep} ep_{id} \xrightarrow{ep} ep_{*} \xrightarrow{ep} ep_{*})$
$\qquad\qquad\qquad (\lambda\ p\ r\ \ (x,\ y) \to (fmap2\ p\ r\ x,\ fmap2\ p\ r\ y))$

instance $\mathsf{P\_fmap2}\ \mathsf{EmptyF}$ **where**
$\quad fmap2 \quad = \quad from\ (ep_{id} \xrightarrow{ep} ep_{id} \xrightarrow{ep} ep_{\mathsf{Empty}} \xrightarrow{ep} ep_{\mathsf{Empty}})$
$\qquad\qquad\qquad (\lambda\ p\ r \to id)$

instance $\mathsf{P\_fmap2}\ \mathsf{ParF}$ **where**
$\quad fmap2 \quad = \quad from\ (ep_{id} \xrightarrow{ep} ep_{id} \xrightarrow{ep} ep_{\mathsf{Par}} \xrightarrow{ep} ep_{\mathsf{Par}})$
$\qquad\qquad\qquad (\lambda\ p\ r \to p)$

instance $\mathsf{P\_fmap2}\ \mathsf{RecF}$ **where**
$\quad fmap2 \quad = \quad from\ (ep_{id} \xrightarrow{ep} ep_{id} \xrightarrow{ep} ep_{\mathsf{Rec}} \xrightarrow{ep} ep_{\mathsf{Rec}})$
$\qquad\qquad\qquad (\lambda\ p\ r \to r)$

instance $(\mathsf{FunctorOf}\ h\ d, \mathsf{P\_fmap2}\ h, \mathsf{P\_fmap2}\ g)$
$\quad \Rightarrow \mathsf{P\_fmap2}\ (\mathsf{CompF}\ d\ g)$ **where**
$\quad fmap2 \quad = \quad from\ (ep_{id} \xrightarrow{ep} ep_{id} \xrightarrow{ep} ep_{@} \xrightarrow{ep} ep_{@})$
$\qquad\qquad\qquad (\lambda\ p\ r \to pmap\ (fmap2\ p\ r))$

instance $\mathsf{P\_fmap2}\ (\mathsf{ConstF}\ t)$ **where**
$\quad fmap2 \quad = \quad from\ (ep_{id} \xrightarrow{ep} ep_{id} \xrightarrow{ep} ep_{\mathsf{Const}} \xrightarrow{ep} ep_{\mathsf{Const}})$
$\qquad\qquad\qquad (\lambda\ p\ r \to id)$

Figure 3.7: Compiled *fmap*2 function

$$
\begin{array}{lcl}
ep_{+} & :: & \mathsf{EP}\ (\mathsf{SumF}\ g\ h\ p\ r)\ (\mathsf{Either}\ (g\ p\ r)\ (h\ p\ r)) \\
ep_{+} & = & \mathsf{EP}\ unSumF\ (either\ \mathsf{InL}\ \mathsf{InR}) \\
\\
ep_{*} & :: & \mathsf{EP}\ (\mathsf{ProdF}\ g\ h\ p\ r)\ \ (g\ p\ r,\ h\ p\ r) \\
ep_{*} & = & \mathsf{EP}\ unProdF\ (uncurry\ (:\!*\!:)) \\
\\
ep_{\mathsf{Empty}} & :: & \mathsf{EP}\ (\mathsf{EmptyF}\ p\ r)\ () \\
ep_{\mathsf{Empty}} & = & \mathsf{EP}\ unEmptyF\ (const\ \mathsf{Empty}) \\
\\
ep_{\mathsf{Par}} & :: & \mathsf{EP}\ (\mathsf{ParF}\ p\ r)\ p \\
ep_{\mathsf{Par}} & = & \mathsf{EP}\ unParF\ \mathsf{Par} \\
\\
ep_{\mathsf{Rec}} & :: & \mathsf{EP}\ (\mathsf{RecF}\ p\ r)\ r \\
ep_{\mathsf{Rec}} & = & \mathsf{EP}\ unRecF\ \mathsf{Rec} \\
\\
ep_{@} & :: & \mathsf{EP}\ (\mathsf{CompF}\ d\ g\ p\ r)\ (d\ (g\ p\ r)) \\
ep_{@} & = & \mathsf{EP}\ unCompF\ \mathsf{Comp} \\
\\
ep_{\mathsf{Const}} & :: & \mathsf{EP}\ (\mathsf{ConstF}\ t\ p\ r)\ t \\
ep_{\mathsf{Const}} & = & \mathsf{EP}\ unConstF\ \mathsf{Const}
\end{array}
$$

Figure 3.8: Embedding-projection pairs between datatype and type synonym representations of pattern functors

$$
\begin{array}{lcl}
\multicolumn{3}{l}{genEP \in Name \times Code \times Type \rightarrow Code} \\
genEP(f, ep, \tau \rightarrow \sigma) & = & genEP(f, ep, \tau) + \text{“}\ \xrightarrow{ep}\ \text{”} + genEP(f, ep, \sigma) \\
genEP(f, ep, f'\ \_\ \_) & & \\
\quad \mid f == f' & = & ep \\
genEP(f, ep, \_) & = & \text{“}ep_{id}\text{”}
\end{array}
$$

Figure 3.9: Generating conversion functions between types using type synonym and datatype pattern functors

the datatype versions and the type synonym versions of the pattern functors. These are defined in figure 3.8. Now in the first branch in our *fmap*2 example the expression

$$\lambda\ p\ r \to \mathit{either}\ (\mathit{fmap}2\ p\ r)\ (\mathit{fmap}2\ p\ r)$$

has type

$$(a \to c) \to (b \to d) \to (\mathsf{Either}\ a\ b \to \mathsf{Either}\ c\ d)$$

but we need something of type

$$(a \to c) \to (b \to d) \to (\mathsf{SumF}\ a\ b \to \mathsf{SumF}\ c\ d)$$

We can get an embedding-projection pair between these types by using ep_{id}, $(\xrightarrow{ep})$ and the embedding-projection pairs from figure 3.8.

$$ep_{id} \xrightarrow{ep} ep_{id} \xrightarrow{ep} (ep_{+} \xrightarrow{ep} ep_{+})$$

The meta function *genEP* defined in figure 3.9 generates these isomorphisms given the name of the polytypic type variable, an isomorphism for the current pattern functor and the type given in the polytypic construct. So to generate the above isomorphism we make the following call

$$\mathit{genEP}(f, ep_{+}, (a \to c) \to (b \to d) \to f\ a\ b \to f\ c\ d)$$

The keen observer will notice that *genEP* actually generates a slightly more complicated isomorphism, but we can acquire the simpler one by rewriting $ep_{id} \xrightarrow{ep} ep_{id}$ as ep_{id} in the isomorphism generated by *genEP*.

Unfortunately we cannot generate these isomorphisms for arbitrary types. With this version of *genEP* we require that the polytypic type variable does not occur as an argument to another type. This is a serious restriction that invalidates for instance the polytypic *zip* function

$$\textbf{polytypic}\ \mathit{fzip} :: (f\ a\ b,\ f\ c\ d) \to \mathsf{Maybe}\ (f\ \ (a,\ c)\ \ (b,\ d))$$

both because of the argument pair and more seriously the Maybe in the result type. To solve the problem in this case we could extend the *genEP* function to handle pairs and Maybe as shown in figure 3.10 (in fact the extended version of *genEP* handles arbitrary instances of the Functor class), but the problem remains for other data types. We could extend it further by giving a generic definition of an embedding-projection pair for a datatype using the definition for sums and products from section 1.3.5. Another, more complete (and vastly more simple) solution is to let the user take care of the problem by requiring that the branches in a polytypic definition uses the data type interpretation instead of the type synonym interpretation of the pattern functors.

$$
\begin{array}{lcll}
genEP & \in & Name \times Code \times Type \to Code & \\
genEP(f, ep, \tau \to \sigma) & = & genEP(f, ep, \tau) + \text{“} \xrightarrow{ep} \text{”} + genEP(f, ep, \sigma) & \\
genEP(f, ep, (\tau, \sigma)) & = & \text{“}ep_{(,)}\ (\text{”} + genEP(f, ep, \tau) + \text{“})\ (\text{”} & \\
 & & \quad + genEP(f, ep, \sigma) + \text{“})\text{”} & \\
genEP(f, ep, d\ \tau) & = & \text{“}ep_{\mathsf{Functor}}\ (\text{”} + genEP(f, ep, \tau) + \text{“})\text{”} & \\
 & & & [\text{if } \mathsf{Functor}\ d] \\
genEP(f, ep, f'\ \_\ \_) & = & ep & [\text{if } f == f'] \\
genEP(f, ep, \_) & = & \text{“}ep_{id}\text{”} &
\end{array}
$$

Figure 3.10: Extending *genEP*

$$
\begin{array}{lcl}
cata & :: & \mathsf{Regular}\ d \Rightarrow (\Phi_d\ a\ b \to b) \to d\ a \to b \\
cata\ f & = & f \circ fmap2\ id\ (cata\ f) \circ out \\
 & & \\
pmap & :: & \mathsf{Regular}\ d \Rightarrow (a \to b) \to d\ a \to d\ b \\
pmap\ f & = & inn \circ fmap2\ f\ (pmap\ f) \circ out
\end{array}
$$

Figure 3.11: Definitions of *cata* and *pmap*

3.4 Translating types

As we have seen, with the class translation calls to polytypic functions look the same way in the compiled code as they do in the PolyP code. The only thing that we need to do is to infer the new type. An advantage when inferring this new type is that all types have already been inferred in PolyP's type system. The only thing we have to do is to adapt the types to fit in our framework. Furthermore we only have to change the types of polytypic functions (that is functions that uses polytypic functions without instantiating them, such as *pmap* here). As we saw in section 3.3 we also have to infer class constraints in the instance declarations of our polytypic classes.

We begin with an example to illustrate the method. Suppose we have the two functions *cata* and *pmap* from figure 3.11 together with *fmap*2 from figure 3.6. We start by creating a table of all the top level functions and their (preliminary) types as given by the PolyP type inference algorithm. In this case we would get the following table.

$$
\begin{array}{lcl}
inn & :: & \mathsf{FunctorOf}\ f\ d \Rightarrow f\ a\ (d\ a) \to d\ a \\
out & :: & \mathsf{FunctorOf}\ f\ d \Rightarrow d\ a \to f\ a\ (d\ a) \\
fmap2 & :: & \mathsf{P\_fmap2}\ f \Rightarrow (a \to c) \to (b \to d) \to f\ a\ b \to f\ c\ d \\
pmap & :: & \mathsf{FunctorOf}\ f\ d \Rightarrow (a \to b) \to d\ a \to d\ b \\
cata & :: & \mathsf{FunctorOf}\ f\ d \Rightarrow (f\ a\ b \to b) \to d\ a \to b
\end{array}
$$

In case of polytypic functions we have to translate the PolyP type into a Haskell type. This is done by replacing the constraint Regular d by FunctorOf f d for a free type variable f and consequently replacing Φ_d by this f. Functions defined by the **polytypic** construct get a constraint saying that the functor must have

$$
\begin{array}{lcl}
[\![\_]\!] & :: & \mathit{PolyPTypeDecl} \to \mathit{TypeDecl} \\
[\![\textbf{polytypic}\ p :: \mathsf{Poly}\ f \Rightarrow \tau]\!] & = & \mathsf{P\_}p\ f \Rightarrow \tau \\
[\![v :: \mathsf{Regular}\ d \Rightarrow \tau]\!] & = & v :: \mathsf{FunctorOf}\ f\ d \Rightarrow \tau[f/\Phi_d] \\
[\![v :: \mathsf{Poly}\ f \Rightarrow \tau]\!] & = & v :: \tau \\
[\![v :: \tau]\!] & = & v :: \tau
\end{array}
$$

Figure 3.12: Translation of PolyP type declaraions into Haskell

an instance of the corresponding class. In the case of *inn*, *out* and *fmap*2 this translation yields the correct type, but for *pmap* and *cata* the types have to be constrained further.

Using this table we reinfer the types of all the top level functions, updating the table when necessary. In the case of mutually recursive polytypic functions we may have to iterate the inference a few times to get the correct type.

In our example, the use of *fmap*2 in both *cata* and *pmap* adds the constraint P_fmap2 f to the types rendering the following final types.

$$
\begin{array}{lcl}
\mathit{cata} & :: & (\mathsf{FunctorOf}\ f\ d, \mathsf{P\_fmap2}\ f) \Rightarrow (f\ a\ b \to b) \to d\ a \to b \\
\mathit{pmap} & :: & (\mathsf{FunctorOf}\ f\ d, \mathsf{P\_fmap2}\ f) \Rightarrow (a \to b) \to d\ a \to d\ b
\end{array}
$$

When inferring the types of the instance declarations we are only interested in the class constraints (we already know what the type should be), but the easiest way of inferring these is to infer the type in the same way as for the functions and discard the type body.

After this simple example we can describe the type inference more formally. The initial table of the types of top level functions is obtained by translating the types inferred by the PolyP type inference into Haskell types. This translation function is defined in figure 3.12. Note that the translation actually translates type declarations into type declarations. The first equation of the definition states that a declaration of the type of a function p defined using the **polytypic** keyword is translated into a type declaration of the function p with the extra constraint P_p f. The constraint Poly f in the PolyP type is added by the PolyP compiler to indicate which type variable the function pattern matches over. The second equation states that d is regular datatype in the PolyP setting if there is an instance FunctorOf f d for some pattern functor f. We also substitute f for occurrences of Φ_d in the PolyP type. The third equation treats cases where we have a polytypic function that operates on pattern functors and not on regular datatypes. For instance we might define

$$
\begin{array}{l}
\mathit{fsum} :: \mathsf{Poly}\ f \Rightarrow f\ \mathsf{Int}\ \mathsf{Int} \to \mathsf{Int} \\
\mathit{fsum} = \mathit{fcrush}\ (+)\ 0
\end{array}
$$

where

$$
\begin{array}{l}
\textbf{polytypic}\ \mathit{fcrush} :: \mathsf{Poly}\ f \Rightarrow (a \to a \to a) \to a \to f\ a\ a \to a \\
\quad = \ldots
\end{array}
$$

$$\frac{}{\mathit{update}\ \Omega\ \emptyset \Rightarrow \Omega}$$

$$\frac{\Omega \cdot x : \tau, \emptyset \vdash e : \tau' \quad \mathit{update}\ (\Omega \cdot x : \tau')\ \Sigma \Rightarrow \Omega'}{\mathit{update}\ (\Omega \cdot x : \tau)\ (\Sigma \cdot x = e) \Rightarrow \Omega'}$$

$$\frac{\mathit{update}\ \Omega\ \Sigma \Rightarrow \Omega}{\mathit{iter}\ \Omega\ \Sigma \Rightarrow \Omega} \qquad \frac{\mathit{update}\ \Omega\ \Sigma \Rightarrow \Omega' \quad \mathit{iter}\ \Omega'\ \Sigma \Rightarrow \Omega''}{\mathit{iter}\ \Omega\ \Sigma \Rightarrow \Omega''}$$

$$\frac{}{\mathit{infer}\ \Omega\ \emptyset \Rightarrow \Omega}$$

$$\frac{\begin{array}{c}\mathit{iter}\ (\Omega \cdot x_1 : [\![\rho_1]\!] \cdot \ldots \cdot x_n : [\![\rho_n]\!])\ (x_1 = e_1 \cdot \ldots \cdot x_n = e_n) \Rightarrow \Omega' \\ \mathit{infer}\ \Omega'\ \mathit{eqns} \Rightarrow \Omega''\end{array}}{\mathit{infer}\ \Omega\ (\{x_1 :: \rho_1 = e_1, \ldots, x_n :: \rho_n = e_n\} : \mathit{eqns}) \Rightarrow \Omega''}$$

Figure 3.13: Inferring the top-level types

In this case, we just remove the Poly f constraint. When analyzing the definition of *fsum* the type inference adds the constraint P_fcrush f.

The top-level type inference algorithm is described in figure 3.13. The function *update* infers the types of all functions in a mutually recursive group and updates the table. The judgement $\Omega, \Gamma \vdash e : \tau$ means that for a table of top-level types Ω and a type environment Γ the expression e has the type τ. The function *iter* takes a top-level table Ω and a group of mutually recursive definitions Σ and iterates *update* until no more updates are made to Ω. The function *infer* is the main function that takes an initially empty table of top-level function types, Ω, and a list of sets of mutually recursive function equations, *eqns*. For each set of mutually recursive definitions we translate the PolyP types ρ_i to Haskell types $[\![\rho_i]\!]$, for each function x_i, and add these types to the top-level table. We then use *iter* to apply the type inference iteratively to this group of equations before continuing with the rest of the program.

We also do some ad-hoc contraint simplification exploiting functional dependencies that is not described here.

This type inference algorithm has not yet been implemented. Instead we use a simpler (incomplete) version that uses more of the types inferred by the PolyP compiler.

3.5 Conclusions

In section 3.1 we defined the goal of this method, namely that we should be able to use polytypic functions in ordinary Haskell programs without having to make any changes to the code. As we have seen we come quite close to this goal, the only thing we have to add to the Haskell program is an instance of the class FunctorOf for every user defined datatype. We have shown how this instance can be generated automatically by our compiler. To achieve our goal in full we could add the FunctorOf class to the set of derivable type classes and let the Haskell compiler generate the instance. This also has the advantage that the Haskell programmer can choose which datatypes should have FunctorOf instances.

As we have seen there are a few limitations with the current implementation. The use of datatype versions of the pattern functors, instead of the type synonyms used in the PolyP programs forced us to conversions that were hard to formulate in the general case. A solution to this comprising a small change to the PolyP language was proposed. The current implementation does cheat a little in the type inference, but it can handle all cases that the original PolyP compiler can handle.

Chapter 4

Generic Haskell

4.1 Introduction

Generic Haskell [HJ] is a system for generic programming based on PolyP [Jan00] using ideas by Hinze [Hin00]. The key concept is the observation that polytypic functions posses polykinded types. So, while PolyP only allowed polytypic functions over datatypes of a specific kind, Generic Haskell allows functions that can be applied to datatypes of any kind.

In this chapter we give a brief introduction to the basics of Generic Haskell. Section 4.2 describes the structure of a datatype in Generic Haskell and section 4.3 deals with how to define generic functions.

4.2 The structure of datatypes

Much like in PolyP, generic functions in Generic Haskell are defined by induction over the structure of datatypes. However the structure of a datatype is defined in a slightly different way in Generic Haskell. In PolyP the structure of a datatype was defined by a pattern functor that abstracted over the recursive call, so to the unary datatypes allowed in PolyP we assigned binary pattern

```
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
```

Figure 4.1: Structure types

functors describing their structure. In Generic Haskell we cannot abstract over the recursive call since we allow non-regular datatypes, where the datatype can be called recursively on different arguments or there could be calls between mutually recursive datatypes. Secondly, the pattern functors used in PolyP were defined in a point-free style, but the structure types in Generic Haskell is not. This is because Generic Haskell allows arbitrarily complex datatypes and hence we would need arbitrarily complex structure combinators to be able to define the structure type in a point-free style. An attempt to do so is presented in chapter 5.

The structure types defined by Generic Haskell are shown in figure 4.1. The type a :+: b describes a choice between a and b, a :∗: b is the product of a and b and Unit is the empty product. These three structure types we recognize from PolyP even if they have slightly different definitions, but Generic Haskell also defines two additional structure types that we did not see in PolyP, namely Con and Label. These types are used to represent constructors and labels in Haskell datatypes, something we did not care about in PolyP. Worth to note is that Generic Haskell does not define any structure types for composition or constant types as PolyP did. Those kind of structure types are not necessary when we do not require point-free type expressions.

Computing the structure type of a datatype is very straight-forward so we just give a few examples.

```
data Nat      =  Zero | Succ Nat
type Nat°     =  Con Unit :+: Con Nat

data List a   =  Nil | Cons a (List a)
type List° a  =  Con Unit :+: Con (a :*: List a)

data Fix f    =  In {out :: f (Fix f)}
type Fix° f   =  Con (Label (f (Fix f)))
```

Note that the structure types use the real types in the recursive calls to avoid getting infinite type synonyms.

4.3 Defining generic functions

With the structure types defined we are ready to define generic functions. As mentioned earlier a generic function over datatypes of arbitrary kind has a polykinded type. That is the type of the generic function is dependent on the kind of the datatype it is applied to. So a generic function in Generic Haskell consists of two things; a kind indexed type (described in section 4.3.1), and a type index value (described in section 4.3.2).

$$\begin{array}{lcl}
\mathit{Map}\,\{\!\![\star]\!\!\}\ t_1\ t_2 & = & t_1 \to t_2 \\
\mathit{Map}\,\{\!\![\kappa_1 \to \kappa_2]\!\!\}\ t_1\ t_2 & = & \\
\quad \forall u_1\, u_2.\ \mathit{Map}\,\{\!\![\kappa_1]\!\!\}\ u_1\ u_2 \to \mathit{Map}\,\{\!\![\kappa_2]\!\!\}\ (t_1\ u_1)\ (t_2\ u_2) & &
\end{array}$$

Figure 4.2: The type of a generic map function

$$\begin{array}{lllcl}
\mathit{gmap}\ \{\!|t :: \kappa|\!\} & & & :: & \mathit{Map}\ \{\!\![\kappa]\!\!\}\ t\ t \\
\mathit{gmap}\ \{\!|{:}{+}{:}|\!\} & mA\ mB & (\mathsf{Inl}\ x) & = & \mathsf{Inl}\ (mA\ x) \\
\mathit{gmap}\ \{\!|{:}{+}{:}|\!\} & mA\ mB & (\mathsf{Inr}\ y) & = & \mathsf{Inr}\ (mB\ y) \\
\mathit{gmap}\ \{\!|\mathsf{Con}\ c|\!\} & mA & (\mathsf{Con}\ x) & = & \mathsf{Con}\ (mA\ x) \\
\mathit{gmap}\ \{\!|\mathsf{Unit}|\!\} & & \mathsf{Unit} & = & \mathsf{Unit} \\
\mathit{gmap}\ \{\!|{:}{*}{:}|\!\} & mA\ mB & (x\ {:}{*}{:}\ y) & = & mA\ x\ {:}{*}{:}\ mB\ y \\
\mathit{gmap}\ \{\!|\mathsf{Label}\ l|\!\} & mA & (\mathsf{Label}\ x) & = & \mathsf{Label}\ (mA\ x) \\
\mathit{gmap}\ \{\!|\mathsf{Int}|\!\} & & n & = & n \\
\mathit{gmap}\ \{\!|\mathsf{Char}|\!\} & & c & = & c
\end{array}$$

Figure 4.3: Generic map

4.3.1 Kind indexed types

A kind indexed type is a type defined by induction over a kind. To illustrate this we look at a specific example, namely the generic map function whose kind indexed type is defined in figure 4.2. This type states that a map function from t_1 to t_2 of kind $\star$ is a function from t_1 to t_2, and a map function from t_1 to t_2 of kind $\kappa_1 \to \kappa_2$ is a function that for any u_1 and u_2 of kind κ_1, given a map between u_1 and u_2 constructs a map between $t_1\ u_1$ and $t_2\ u_2$. To get the familiar map over lists we apply *Map* to the list datatype constructor as follows

$$\mathit{Map}\,\{\!\![\star \to \star]\!\!\}\ []\ \ [] = \forall a\, b.\ (a \to b) \to [a] \to [b]$$

Kind arguments are always enclosed in $\{\!\![$funny brackets$]\!\!\}$.

4.3.2 Type indexed values

Analogously to a kind indexed type, a type indexed value is defined by induction over a type (or rather a type structure). Figure 4.3 defines the generic map function whose kind indexed type was defined in the previous section. The type declaration states that the function *gmap* for a type t of kind k is a mapping function from t to t, so

$$\begin{array}{lcl}
\mathit{gmap}\,\{\!|[]|\!\} & :: & \mathit{Map}\,\{\!\![\star \to \star]\!\!\}\ []\ \ [] \\
 & = & \forall a\, b.\ (a \to b) \to [a] \to [b]
\end{array}$$

Which is exactly what we want. Note that type arguments are enclosed in $\{\!|$different funny brackets$|\!\}$. When defining *gmap* we have to give definitions for each of the structure types defined in section 4.2 as well as for primitive types. Note that the structure types Con and Label has arguments c and l respectively when passed as a type argument. These arguments contain information about the constructor or label involved, such as name, fixity and arity.

Each equation of the definition has a type matching the type declaration of $gmap$, so the type of $gmap \{\!| {:}{+}{:} |\!\}$ is $Map \{\![\star \to \star \to \star]\!\}\ ({:}{+}{:})\ ({:}{+}{:})$ and so on.

The structure types of section 4.2 could also contain (top level) abstraction, variables and application. These cases are defined by the Generic Haskell compiler and cannot be overridden. In chapter 7 we describe these definitions in more detail.

Chapter 5

Class translation for Generic Haskell

5.1 Introduction

In chapter 3 we saw how we could use the class translation to implement PolyP programs in a way that allowed us to use polytypic functions in ordinary Haskell code and enabled separate compilation of generic functions. It would be very nice if we could do the same thing for Generic Haskell. This chapter describes an attempt to do this based on a sketch by deWit [dW99]. As we shall see this attempt is not the brilliant success we experienced when doing the class translation for PolyP.

When we were dealing with PolyP we only considered regular, unary datatypes, something that made life quite simple. When moving to Generic Haskell we have to consider non-regular datatypes of arbitrary kind. The first thing to note is that we at least have to specialize the generic functions to particular kinds, since the type of a generic function depends on the kind of the type argument. Take for example the generic map function defined in figure 4.3. There is no way we

$$\begin{array}{lcll}
[\![\star]\!] & = & 0 & \\
[\![\star \to \kappa]\!] & = & n+1 & \text{if } [\![\kappa]\!] = n < 9 \\
[\![\kappa \to \nu]\!] & = & \kappa\nu & \\
 & & & \\
[\![(\star \to \star) \to \star \to \star]\!] & = & 11 & \\
[\![((\star \to \star) \to \star) \to \star]\!] & = & (10)0 &
\end{array}$$

Figure 5.1: Abbreviated kind notation

could use the same name for *gmap* instances at different kinds.

$$\begin{aligned} &gmap\ \{\!|t :: \star \to \star|\!\} &&:: \quad (a \to b) \to t\ a \to t\ b \\ &gmap\ \{\!|t :: \star|\!\} &&:: \quad t \to t \end{aligned}$$

Instead we try to generate a class for each kind the generic function is used at. So in the case of *gmap* we would get the following classes:

```
class G_gmap_0 t where
  gmap_0  ::  t → t

class G_gmap_1 t where
  gmap_1  ::  (a → b) → t a → t b

class G_gmap_2 t where
  gmap_2  ::  (a → c) → (b → d) → t a b → t c d
⋮
```

We index the class and the corresponding member function with the kind at which the function is applied. To save some space we introduce a shorter notation for kinds, described in figure 5.1.

5.2 Pattern functors

As we saw in chapter 4 every datatype has a structure type associated with it. For instance, the structure type of the list datatype is

```
type List° a  =  Con Unit :+: Con (a :*: [a])
```

The problem with this structure type is that it is not in a point-free style. Since we want to build up an instance of G_gmap for List° from instances for the pattern functors we must express List° in a point-free style.

To be able to express an arbitrary type expression in a point-free style we need a combinator basis for our type language. In PolyP (chapter 2) we had a very restricted type language which meant that we only needed a few (monomorphic) combinators (see figure 2.1). In Generic Haskell on the other hand the type language is a restricted version of simply typed lambda calculus extended with binary sums, nullary and binary products, function types, constructor types and label types (see section 4.2). To express the extensions to the lambda calculus we use the structure types defined in figure 4.1, the remaining task is then to define a suitable combinator basis for the lambda calculus. A valid choice would be the S, K and I combinators but since we want the expressions to be as simple as possible we replace the S combinator with the four combinators C11, C21, C12 and C22. We also rename I to Id and K to Exl and add Exr for symmetry reasons.

$$
\begin{array}{lcl}
\mathsf{Id} & :: & \forall\kappa.\kappa\kappa \\
\mathsf{Id}\ a & = & a \\
\\
\mathsf{Exl} & :: & \forall\kappa\nu.\kappa\nu\kappa \\
\mathsf{Exl}\ a\ b & = & a \\
\\
\mathsf{Exr} & :: & \forall\kappa\nu.\kappa\nu\nu \\
\mathsf{Exr}\ a\ b & = & b \\
\\
\mathsf{C11} & :: & \forall\kappa\nu\mu.(\kappa\mu)(\nu\kappa)\nu\mu \\
\mathsf{C11}\ a\ b\ c & = & a\ (b\ c) \\
\\
\mathsf{C21} & :: & \forall\kappa\nu\mu\pi.(\kappa\nu\pi)(\mu\kappa)(\mu\nu)\mu\pi \\
\mathsf{C21}\ a\ b\ c\ d & = & a\ (b\ d)\ (c\ d) \\
\\
\mathsf{C12} & :: & \forall\kappa\nu\mu\pi.(\kappa\pi)(\nu\mu\pi)\nu\mu\pi \\
\mathsf{C12}\ a\ b\ c\ d & = & a\ (b\ c\ d) \\
\\
\mathsf{C22} & :: & \forall\kappa\nu\mu\pi\rho.(\kappa\nu\rho)(\mu\pi\kappa)(\mu\pi\nu)\mu\pi\rho \\
\mathsf{C22}\ a\ b\ c\ d\ e & = & a\ (b\ d\ e)\ (c\ d\ e)
\end{array}
$$

Figure 5.2: Generalized pattern functors

The complete set of combinators is shown in figure 5.2. These combinators form a basis for lambda calculus since we have that

$$
\begin{array}{lcl}
\mathsf{C21}\ \mathsf{Id} & = & \lambda xyz.\mathsf{Id}(xz)(yz) \\
 & = & \lambda xyz.xz(yz) \\
 & = & \mathsf{S}
\end{array}
$$

The rules for transforming types to a point-free style is shown in figure 5.4. In the rules x denotes a variable, τ denotes an arbitrary type expression and $FV(\tau)$ denotes the set of free variables in τ. The rules have been divided into three categories: E-rules, L-rules and R-rules.

$$
\begin{array}{lcl}
\mathsf{Exl}\ \mathsf{Id} & = & (\lambda\ a\ b.\ a)\ \mathsf{Id} \\
 & = & \lambda\ b.\mathsf{Id} \\
 & = & \lambda\ b\ a.\ a \\
 & = & \mathsf{Exr} \\
\mathsf{C11}\ (\mathsf{C11}\ \tau) & = & \lambda\ a\ b.\ \mathsf{C11}\ \tau\ (a\ b) \\
 & = & \lambda\ a\ b\ c.\ \tau\ (a\ b\ c) \\
 & = & \mathsf{C12}\ \tau \\
\mathsf{C21}\ (\mathsf{C21}\ \tau) & = & \lambda\ a\ b\ c.\ \mathsf{C21}\ \tau\ (a\ c)\ (b\ c) \\
 & = & \lambda\ a\ b\ c\ d.\ \tau\ (a\ c\ d)\ (b\ c\ d) \\
 & = & \mathsf{C22}\ \tau
\end{array}
$$

Figure 5.3: Some combinator equalities

$$\frac{\tau \longrightarrow \tau'}{\lambda x.\tau \longrightarrow \lambda x.\tau'} \qquad (\mathrm{E}_\lambda)$$

$$\frac{\tau_1 \longrightarrow \tau_1'}{\tau_1\ \tau_2 \longrightarrow \tau_1'\ \tau_2} \qquad (\mathrm{E}_{\mathrm{app}}.1)$$

$$\frac{\tau_2 \longrightarrow \tau_2'}{\tau_1\ \tau_2 \longrightarrow \tau_1\ \tau_2'} \qquad (\mathrm{E}_{\mathrm{app}}.2)$$

$$\frac{}{\lambda x.\tau \longrightarrow \mathsf{Exl}\ \tau}\ x \notin FV(\tau) \qquad (\mathrm{L}_{\mathrm{exl}})$$

$$\frac{}{\lambda x.x \longrightarrow \mathsf{Id}} \qquad (\mathrm{L}_{\mathrm{id}})$$

$$\frac{}{\lambda x.\tau\ x \longrightarrow \tau}\ x \notin FV(\tau) \qquad (\mathrm{L}_\eta)$$

$$\frac{}{\lambda x.\tau_1\ \tau_2 \longrightarrow \mathsf{C11}\ \tau_1\ (\lambda x.\tau_2)}\ x \notin FV(\tau_1) \qquad (\mathrm{L}_{\mathrm{c11}})$$

$$\frac{}{\lambda x.\tau_1\ \tau_2\ \tau_3 \longrightarrow \mathsf{C21}\ \tau_1\ (\lambda x.\tau_2)\ (\lambda x.\tau_3)}\ x \notin FV(\tau_1) \qquad (\mathrm{L}_{\mathrm{c21}}.1)$$

$$\frac{}{\lambda x.\tau_1\ \tau_2 \longrightarrow \mathsf{C21}\ \mathsf{Id}\ (\lambda x.\tau_1)\ (\lambda x.\tau_2)}\ x \in FV(\tau_1) \qquad (\mathrm{L}_{\mathrm{c21}}.2)$$

$$\frac{}{\mathsf{Exl}\ \mathsf{Id} \longrightarrow \mathsf{Exr}} \qquad (\mathrm{R}_{\mathrm{exr}})$$

$$\frac{}{\mathsf{C11}\ (\mathsf{C11}\ \tau) \longrightarrow \mathsf{C12}\ \tau} \qquad (\mathrm{R}_{\mathrm{c12}})$$

$$\frac{}{\mathsf{C21}\ (\mathsf{C21}\ \tau) \longrightarrow \mathsf{C22}\ \tau} \qquad (\mathrm{R}_{\mathrm{c22}})$$

Figure 5.4: Point-free rules

The E-rules are the general rules stating that we can transform any subexpression of an expression. The L-rules deal with how to remove lambda abstractions. Remember that we could write $\mathsf{S} = \mathsf{C21\ Id}$. Using our rules we can derive this as follows:

$$\begin{array}{rcl}
\mathsf{S} & = & \lambda xyz.xz(yz) \\
(\mathrm{L}_{\mathrm{c21}}.2) & \longrightarrow & \lambda xy.\ \mathsf{C21\ Id}\ (\lambda z.xz)\ (\lambda z.yz) \\
(\mathrm{L}_{\eta})^{*} & \longrightarrow_{*} & \mathsf{C21\ Id}
\end{array}$$

Finally the R-rules simplifies the type expression by introducing the combinators Exr, C12 and C22 using the equalities from figure 5.3.

Using these rules we can derive the point-free form of the structure type for the list data type shown in the beginning of this section:

$$\begin{array}{rcl}
\mathsf{List}^{*} & = & \lambda a.\ \mathsf{Con\ Unit}\ {:}{+}{:}\ \mathsf{Con}\ (a\ {:}{*}{:}\ [a]) \\
(\mathrm{L}_{\mathrm{c11}}) & \longrightarrow & \mathsf{C11}\ ((:+:)\ (\mathsf{Con\ Unit}))\ (\lambda a.\ \mathsf{Con}\ (a\ {:}{*}{:}\ [a])) \\
(\mathrm{L}_{\mathrm{c11}}) & \longrightarrow & \mathsf{C11}\ ((:+:)\ (\mathsf{Con\ Unit}))\ (\mathsf{C11\ Con}\ (\lambda a.\ a\ {:}{*}{:}\ [a]) \\
(\mathrm{L}_{\mathrm{c21}}.1) & \longrightarrow & \mathsf{C11}\ ((:+:)\ (\mathsf{Con\ Unit}))\ (\mathsf{C11\ Con}\ (\mathsf{C21}\ (:*:)\ (\lambda a.a)\ (\lambda a.[a]))) \\
(\mathrm{L}_{\mathrm{id}}) & \longrightarrow & \mathsf{C11}\ ((:+:)\ (\mathsf{Con\ Unit}))\ (\mathsf{C11\ Con}\ (\mathsf{C21}\ (:*:)\ \mathsf{Id}\ (\lambda a.[a]))) \\
(\mathrm{L}_{\eta}) & \longrightarrow & \mathsf{C11}\ ((:+:)\ (\mathsf{Con\ Unit}))\ (\mathsf{C11\ Con}\ (\mathsf{C21}\ (:*:)\ \mathsf{Id}\ []))
\end{array}$$

5.2.1 Kind polymorphism

Since Haskell's class system does not allow kind polymorphism we cannot define the combinators from figure 5.2 directly. Instead we must define a separate data type for each instance the combinator is used at. Let $T :: \forall v_1 \ldots v_n.\kappa$ be a polymorphic type, then $T[\kappa_1, \ldots, \kappa_n] = \kappa[\kappa_1/v_1 \ldots \kappa_n/v_n]$ denotes an instance of T where v_i has been instantiated to κ_i. Now if we return to the list example from the previous section we can define the structure type for lists using the correct instances of our polymorphic combinators:

$$\begin{array}{l}
\textbf{type}\ \mathsf{List}^{*} = \mathsf{C11}[0,0,0]\ ((:+:)\ (\mathsf{Con\ Unit})) \\
\qquad\qquad\qquad\qquad (\mathsf{C11}[0,0,0]\ \mathsf{Con}\ (\mathsf{C21}[0,0,0,0]\ (:*:)\ \mathsf{Id}[0]\ []))
\end{array}$$

In this example all kind variables are instantiated to $\star$, so in order to define this type in Haskell we would need to define the following types:

$$\begin{array}{lcl}
\textbf{newtype}\ \mathsf{C11}[0,0,0]\ a\ b\ c & = & \mathsf{C11}[0,0,0]\ (a\ (b\ c)) \\
\textbf{newtype}\ \mathsf{C21}[0,0,0,0]\ a\ b\ c\ d & = & \mathsf{C21}[0,0,0,0]\ (a\ (b\ d)\ (c\ d)) \\
\textbf{newtype}\ \mathsf{Id}[0]\ a & = & \mathsf{Id}[0]\ a
\end{array}$$

In the following we drop the kind indices when all kind variables are instantiated to $\star$. Thus we write C11 for $\mathsf{C11}[0,0,0]$ and equivalently for the other combinators.

$$
\begin{array}{lll}
\textbf{class}\ \mathsf{StructureOf_0}\ s\ d \mid d \to s\ \textbf{where} & & \\
\quad inn_0 & :: & s \to d \\
\quad out_0 & :: & d \to s \\
\\
\textbf{class}\ \mathsf{StructureOf_1}\ s\ d \mid d \to s\ \textbf{where} & & \\
\quad inn_1 & :: & s\ a \to d\ a \\
\quad out_1 & :: & d\ a \to s\ a \\
\\
\textbf{class}\ \mathsf{StructureOf_{10}}\ s\ d \mid d \to s\ \textbf{where} & & \\
\quad inn_{10} & :: & s\ (a :: \star \to \star) \to d\ a \\
\quad out_{10} & :: & d\ (a :: \star \to \star) \to s\ a
\end{array}
$$

Figure 5.5: StructureOf class definitions

For most datatypes of kind $\star$, $\star \to \star$ or $\star \to \star \to \star$ we can find a structure type that only uses combinators where all kind variables have been instantiated to $\star$ (but we would need to refine the rules from figure 5.4), however as soon as we have datatypes with higher order kinds or with more that two arguments we have to use other instances of the combinators. For example the structure type of the fix point datatype is

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{Fix}\ f & = & \mathsf{In}\ (f\ (\mathsf{Fix}\ f)) \\
\textbf{type}\ \mathsf{Fix}^* & = & \mathsf{C11}[0,1,0]\ \mathsf{Con}\ (\mathsf{C21}[1,0,1,0]\ \mathsf{Id}[1]\ \mathsf{Id}[1]\ \mathsf{Fix})
\end{array}
$$

At this point we start to realize that it might not be such a good idea to apply the class translation method to Generic Haskell. Since we can define arbitrarily complex datatypes, we might need arbitrarily complex instances of our combinators and thus we cannot define them in advance as we did in the class translation for PolyP (see chapter 3). On the other hand there might be a reasonably small set of combinators that can express most common datatypes, so hopefully our efforts are not totally in vain.

5.3 The StructureOf classes

Once we have computed the structure type of a specific datatype we want to link the two together. In the class translation for PolyP this was done by the FunctorOf class (section 3.2). Since Generic Haskell deals with datatypes of arbitrary kind we cannot use a single class but instead we need one class for each kind. The definitions of StructureOf for kinds $\star$, $\star \to \star$ and $(\star \to \star) \to \star$ are shown in figure 5.5. Note that we have to make use of explicit kind annotation in the definition of $\mathsf{StructureOf_{10}}$ to get the kinds right. An important difference between the structure types of Generic Haskell and the pattern functors from PolyP is that a structure type has the same kind as their corresponding datatypes while the pattern functors takes an extra argument representing the recursive call. This is reflected in the definitions of the StructureOf classes.

```
class StructureOf₁ List* []
    inn₁ (C11 (Inl (Con Unit)))                          = []
    inn₁ (C11 (Inr (C11 (Con (C21 (Id x :*: xs))))))   = x : xs
    out₁ []         = C11 (Inl (Con Unit))
    out₁ (x : xs)   = C11 (Inr (C11 (Con (C21 (Id x :*: xs)))))
```

Figure 5.6: StructureOf instance for the list datatype

In most cases, creating the instances of the appropriate StructureOf class is very straight-forward (an example is shown in figure 5.6), but in some cases we get the same problem described in section 3.2, that we need to get inside a datatype to do the conversion to and from the structure type. Take for instance the datatype Foo defined by

```
data Foo a = Foo (T a)
```

for some datatype T. A possible (but suboptimal) structure type for Foo is

```
type Foo* = C11 Con (C11 T Id)
```

Now the when we try to define an instance of $\mathsf{StructureOf}_1$ for this type we run into trouble

```
instance StructureOf₁ Foo* Foo where
   inn₁ (C11 (Con (C11 x))) = Foo ?₁
   out₁ (Foo y) = C11 (Con (C11 ?₂))
```

In this case $x, ?_2 :: \mathsf{T}\ (\mathsf{Id}\ a)$ and $y, ?_1 :: \mathsf{T}\ a$, so we would like to have

$$
\begin{aligned}
?_1 &= \mathit{mapT}\ \mathit{unId}\ x \\
?_2 &= \mathit{mapT}\ \mathsf{Id}\ y
\end{aligned}
$$

where *mapT* is a map function for T. In this case we could have avoided the problem by choosing a simpler structure type for Foo but in general this is not possible. The solution is to define the generic map function *gmap* and use it for *mapT*.

5.4 Classes

Similarly to the class translation in PolyP we want to break down a generic function into a class and instance declarations corresponding to the different cases in the definition. Since generic (or type-indexed) functions have kind-indexed types we need one class for each kind. We take as an example the

```
class G_gmap₀ t where
  gmap₀    ::  t → t

class G_gmap₁ t where
  gmap₁    ::  (a → b) → t a → t b

class G_gmap₂ t where
  gmap₂    ::  (a → b) → (c → d) → t a c → t b d

class G_gmap₁₀ t where
  gmap₁₀   ::  (∀ a b. (a → b) → u a → v b) → t u → t v
```

Figure 5.7: *gmap* classes

```
instance G_gmap₂ (:+:) where
  gmap₂ mA mB (Inl x)     =  Inl (mA x)
  gmap₂ mA mB (Inr y)     =  Inr (mB y)

instance G_gmap₁ Con where
  gmap₁ mA (Con x)        =  Con (mA x)

instance G_gmap₀ Unit where
  gmap₀ Unit              =  Unit

instance G_gmap₂ (:*:) where
  gmap₂ mA mB (x :*: y)   =  mA x :*: mB y

instance G_gmap₁ Label where
  gmap₁ mA (Label x)      =  Label (mA x)

instance G_gmap₀ Int where
  gmap₀ n                 =  n

instance G_gmap₀ Char where
  gmap₀ c                 =  c
```

Figure 5.8: Instances for *gmap*

instance (G_gmap$_1$ f, G_gmap$_1$ g) $\Rightarrow$ G_gmap$_1$ (C11 f g) **where**
  $gmap_1\ mA$ (C11 x) $=$ C11 ($gmap_1$ ($gmap_1\ mA$) x)

instance (G_gmap$_2$ f, G_gmap$_1$ g, G_gmap$_1$ h) $\Rightarrow$
  G_gmap$_1$ (C21 f g h) **where**
  $gmap_1\ mA$ (C21 x) $=$ C21 ($gmap_2$ ($gmap_1\ mA$) ($gmap_1\ mA$) x)

instance G_gmap$_1$ Id **where**
  $gmap_1\ mA$ (Id x) $=$ Id ($mA\ x$)

Figure 5.9: G_gmap instances for some combinators

generic function *gmap* defined in figure 4.3. Some of the classes *gmap* translates into are shown in figure 5.7.

Given these classes we create instances of the appropriate class for each of the equations in the generic definition by simply taking the right hand side of the equation to be the right hand side in the instance definition. Note that since we only have one representation of the structure types, and not two as was the case in PolyP (see section 3.3) we don't have to do any conversion to get the right type. Figure 5.8 shows the instance definitions arising from *gmap* function. One troublesome issue with this translation is that we loose the constructor and label descriptors, which means that we cannot translate functions that use this information. We cannot see any simple solutions to this problem.

In the case of PolyP it was enough to provide the instance definitions for the cases given in the **polytypic** construct, but now we also have to define instances for our combinators from section 5.2. Here we run into yet another problem: One of the goals of the class translation was to be able to compile the generic function separately, without any knowledge of how it will be used. But as we saw in the previous section the datatypes on which the generic function is used determine which specializations of the combinators are needed. Unless we can find a reasonably small set of combinators with which we can express most common datatypes, we have gained nothing over the specialization approach that Generic Haskell currently uses.

Even though it is now apparent that this approach of using Haskell's class system to implement Generic Haskell programs is far from practical we finish the list example. Recall the structure type of the standard list datatype from section 5.2:

type List* $=$ C11 ((:+:) (Con Unit)) (C11 Con (C21 (:∗:) Id []))

This type uses the combinators C11, C21 and Id, all specialized to concrete types. The instance definitions of the G_gmap classes for these combinators are given in figure 5.9. Note that it is not always clear which class to make a combinator an instance of. For example we chose to make C11 f g an instance of G_gmap$_1$ provided that f and g are instances of G_gmap$_1$, instead of making

C11 an instance of G_gmap$_{111}$. Had we chosen the latter alternative we would have had to add the following general instances to get the instance we needed:

$$\begin{array}{l}
\mathbf{instance}\ (\mathsf{G\_gmap}_{111}\ f, \mathsf{G\_gmap}_{1}\ g) \Rightarrow \mathsf{G\_gmap}_{11}\ (f\ g)\ \mathbf{where} \\
\quad gmap_{11} \quad = \quad gmap_{111}\ gmap_{1}
\end{array}$$

$$\begin{array}{l}
\mathbf{instance}\ (\mathsf{G\_gmap}_{11}\ f, \mathsf{G\_gmap}_{1}\ g) \Rightarrow \mathsf{G\_gmap}_{1}\ (f\ g)\ \mathbf{where} \\
\quad gmap_{1} \quad = \quad gmap_{11}\ gmap_{1}
\end{array}$$

We need this kind of general instances for the list example even when we choose the former alternative. To be able to use the G_gmap$_1$ instance for C11 we have to have an instance of G_gmap$_1$ for (:+:) (Con Unit). To get this instance we add two general instances

$$\begin{array}{l}
\mathbf{instance}\ (\mathsf{G\_gmap}_{2}\ f, \mathsf{G\_gmap}_{0}\ t) \Rightarrow \mathsf{G\_gmap}_{1}\ (f\ t)\ \mathbf{where} \\
\quad gmap_{1} \quad = \quad gmap_{2}\ gmap_{0}
\end{array}$$

$$\begin{array}{l}
\mathbf{instance}\ (\mathsf{G\_gmap}_{1}\ f, \mathsf{G\_gmap}_{0}\ t) \Rightarrow \mathsf{G\_gmap}_{0}\ (f\ t)\ \mathbf{where} \\
\quad gmap_{0} \quad = \quad gmap_{1}\ gmap_{0}
\end{array}$$

These kind of general instances relies on the extension to GHC allowing overlapping instances. Without this extension we would have to replace the fs in the above instances with (:+:) and Con respectively.

There is a trade-off between having the most general instances, i.e. making C11 an instance of G_gmap$_{111}$, and having reasonably complex kinds. In the example of C11 we probably do not need an instance of G_gmap$_{111}$ so we make do with the simpler instance in figure 5.9.

Finally we need an instance of G_gmap$_1$ for []. It is tempting to add an instance like the following to take care of all datatype of kind $\star \to \star$:

$$\begin{array}{l}
\mathbf{instance}\ (\mathsf{StructureOf}_{1}\ s\ d, \mathsf{G\_gmap}_{1}\ s) \Rightarrow \mathsf{G\_gmap}_{1}\ d\ \mathbf{where} \\
\quad gmap_{1}\ mA \quad = \quad inn_{1} \circ gmap_{1}\ mA \circ out_{1}
\end{array}$$

This instance, however, is not allowed since the instance head is just a type variable. Of course GHC provides a flag to lift this restriction, but for reasons not completely obvious it then causes the type checker to loop (this is probably due to the fact that the prerequisite G_gmap$_1$ s can be unified with G_gmap$_1$ d). What we do instead is to define the instance only for the list datatype.

$$\begin{array}{l}
\mathbf{instance}\ \mathsf{G\_gmap}_{1}\ []\ \mathbf{where} \\
\quad gmap_{1}\ mA \quad = \quad inn_{1} \circ gmap_{1}\ mA \circ out_{1}
\end{array}$$

Note that the body of the instance is exactly the same as the body of the general instance above.

5.5 Conclusions

This attempt at a class translation for Generic Haskell has proven largely unsuccessful. It points at several problems that arises when moving from the regular, fixed kinded datatypes used in PolyP to arbitrary datatypes of arbitrary kind. The most serious problem is that we can no longer use a fixed set of type combinators to express the structure of a datatype. This means that we cannot create all the combinators until we know at which datatypes we want to use generic functions, nor can we compile a generic function without knowing at which datatypes it is called. One of the goals of the class translation was to be able to compile generic functions separately from the code that calls them, something that we have not been able to achieve.

A possible solution to this lack of success would be to find a fixed set of combinators that can express most *common* datatypes. Since most haskell programs does not use higher kinded datatypes, finding a set of combinators that can express datatypes of first order kinds would be quite useful. This is something worth looking into.

Another drawback was that we had to instantiate the classes generated for a generic function explicitly for each datatype, while in PolyP it was enough to define an instance of the FunctorOf class for the datatype to get access to all the polytypic functions. The difference between PolyP and Generic Haskell in this aspect is that in PolyP there is a clear distinction between functions operating on pattern functors and functions operating over real datatypes and it is only the former kind that can be defined using a type case. This avoids the problem since the real datatypes should not be instances of the classes generated from the polytypic definitions. In Generic Haskell on the other hand, the same functions are applied to both structure types and real datatypes and thus we need an instance of the appropriate generated class for every datatype.

We also saw that the passing of constructor and label descriptors used in Generic Haskell is not supported by the class framework used. It might be possible to solve this problem but we have not made any attempts to do so.

In summary we can conclude that the class translation is not really practical for Generic Haskell. However it might be possible to do the class translation if we place (hopefully small) restrictions on which datatypes are allowed.

Chapter 6

Type theory and Alfa

In this and the following chapter we make heavy use of the type theoretic concept of dependent types. We use the Alfa/iAGDA [HR00] system, that is based on Martin-Löf type theory [NPS90] and everything we show here has been implemented in this system. This chapter gives a very brief introduction to dependent types and the Alfa syntax while introducing a few functions and structures that we need further on.

6.1 Dependent types

To have dependent types is to say that types can depend on values. For instance, the type of a function might depend on the value of the functions first argument. A well known example of such a function is the C-function *printf*, that takes a format string, as the first argument, specifying how to print the following arguments. The number of arguments and their types depend on the value of the format string. Of course C does not have a dependent type system and consequently the *printf* function is not strongly typed in C[1] In the following subsections we look at many examples of dependently typed functions in Alfa.

6.2 Introduction to Alfa

6.2.1 An introductory example

The easiest way of getting acquainted with Alfa is to look at examples, so that is what we are going to do. We start by defining our favorite datatype, namely

[1]Some compilers actually does typecheck *printf*, but in those cases the typechecking is tailor-made for the *printf* function and only works if the format string is a value.

the list datatype.

$$\begin{array}{lcl} \mathit{List}\ (A \in \mathit{Set}) & \in & \mathit{Set} \\ \mathit{List}\ A & \equiv & \mathbf{data} \left\{ \begin{array}{l} \mathbf{Nil} \\ \mathbf{Cons}\ (x \in A)(xs \in \mathit{List}\ A) \end{array} \right\} \end{array}$$

There are a few things worth noting here. First note that we use set membership to denote types, $x \in A$ corresponds to $x :: A$ in Haskell. Another thing to note is that the type constructor *List* is also typed: *List* takes a set A as an argument and returns a new set. Just as in Haskell we can choose to write arguments in the left hand side as we did above, but we could equally well have defined *List* as follows

$$\begin{array}{lcl} \mathit{List} & \in & (A \in \mathit{Set}) \to \mathit{Set} \\ \mathit{List} & \equiv & \lambda A.\,\mathbf{data} \left\{ \begin{array}{l} \mathbf{Nil} \\ \mathbf{Cons}\ (x \in A)(xs \in \mathit{List}\ A) \end{array} \right\} \end{array}$$

In this case we could omit the name of the argument and write the type as $\mathit{List} \in \mathit{Set} \to \mathit{Set}$.

The set *Set* is the set of all inductively defined sets, corresponding roughly to the set of concrete Haskell types, or the kind $\star$. This means that we can interpret the type of *List* as the Haskell kind $\star \to \star$. To contain objects such as *Set* and $\mathit{Set} \to \mathit{Set}$ we have the type *Type* (the set of all kinds, using our Haskell analogy). In fact we have an infinite hierarchy of these types $\#0 \in \#1 \in \#2 \in \ldots$, where $\#0 = \mathit{Set}$ and $\#1 = \mathit{Type}$, but we only need *Set* and *Type* here.

The **data** definition is very close to its Haskell counterpart. The only thing worth noting is that the constructor parameters are named. This is to allow dependent parameters, something we will see examples of later on. As illustrated by this example we write constructors in **bold** and other identifiers in *italic*.

Now we can start writing functions over lists. The map function, for instance, can be defined as follows.

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathit{map}\ (A, B \in \mathit{Set}, f \in A \to B, xs \in \mathit{List}\ A) \in \mathit{List}\ B} \\ \mathit{map}\ A\ B\ f\ \mathbf{Nil} & \equiv & \mathbf{Nil} \\ \mathit{map}\ A\ B\ f\ (\mathbf{Cons}\ x\ xs') & \equiv & \mathbf{Cons}\ (f\ x)\ (\mathit{map}\ A\ B\ f\ xs') \end{array}$$

Alfa is completely monomorphic, something that might come as a surprise. This is not a problem, however, since we can use dependent types to simulate parametric polymorphism. Compare the above type to the type of map in a polymorphic language: $\mathit{map} :: \forall\, A\, B.\, (A \to B) \to \mathit{List}\ A \to \mathit{List}\ B$. Instead of quantifying over the types A and B we pass them as parameters to the function. All this explicit passing of types tend to clutter up the code quite a bit, so Alfa provides the ability to hide arguments. Hidden arguments are written within brackets to the right of the type declaration and are not shown in the definition, so if we want to hide the arguments A and B above we would write

$$\begin{array}{lll} \mathit{map}\ (f \in A \to B, \mathit{xs} \in \mathit{List}\ A) \in \mathit{List}\ B & & [A, B \in \mathit{Set}] \\ \mathit{map}\ f\ \mathbf{Nil} & \equiv & \mathbf{Nil} \\ \mathit{map}\ f\ (\mathbf{Cons}\ x\ \mathit{xs}') & \equiv & \mathbf{Cons}\ (f\ x)\ (\mathit{map}\ f\ \mathit{xs}') \end{array}$$

As we can see in the recursive call, hidden arguments do not show up in function calls. We use this feature extensively in this chapter to make the code readable, but we take care only to hide arguments whose values can be inferred from the visible arguments.

6.2.2 Dependent pairs

Our previous example showed how to write ordinary Haskell functions and datatypes in Alfa. Now we look at some examples that you cannot write in Haskell. A dependent pair is a pair in which the type of the second component can depend on the value of the first component. This pair type is sometimes referred to as a disjoint union, since we can regard the first component a tag that decides what type of values can be stored in the second component. In Alfa we can define a dependent pair type as follows:

$$\begin{array}{l} (\times)\ (A \in \mathit{Set}, B \in A \to \mathit{Set}) \in \mathit{Set} \\ A \times B \equiv \mathbf{data}\ \{(\times)\ (a \in A, b \in B\ a)\} \end{array}$$

Note that in the data constructor $(\times)$ the type of b depends on a. In the next chapter we are going to need dependent pairs, but instead of storing elements of sets we want to store elements of types. Therefore we use the following definition instead, where we have replaced *Set* by *Type*.

$$\begin{array}{l} (\times)\ (A \in \mathit{Type}, B \in A \to \mathit{Type}) \in \mathit{Type} \\ A \times B \equiv \mathbf{data}\ \{(\times)\ (a \in A, b \in B\ a)\} \end{array}$$

One thing that might be confusing is that in the type $A \times B$, B is a function taking a value of type A and giving a type, and not a type itself as one might think.

Type theorists would now define an elimination rule for the dependent pairs, stating something like if we can construct an object of type C from an $a \in A$ and a $b \in B\ a$ then we can construct an object of type C from a $p \in A \times B$. Instead of doing this, however, we define two functions *fst* and *snd* to extract the first and second component respectively.

$$\begin{array}{ll} \mathit{fst}\ (p \in A \times B) \in A & [A \in \mathit{Type}, B \in A \to \mathit{Type}] \\ \mathit{fst}\ (a \times b) \equiv a & \end{array}$$

$$\begin{array}{ll} \mathit{snd}\ (p \in A \times B) \in B\ (\mathit{fst}\ p) & [A \in \mathit{Type}, B \in A \to \mathit{Type}] \\ \mathit{snd}\ (a \times b) \equiv b & \end{array}$$

The functions *fst* and *snd* are defined by pattern matching on the pair in the obvious way. Note that we use the function *fst* in the type of *snd*.

6.2.3 Forall quantification

Recall from chapter 4 that we use a lot of universal quantification in our kind indexed types. Since Alfa is completely monomorphic we have to encode this quantification in some way. The way we do this is the way of the constructive mathematician: a proof that $P(x)$ holds for all x is a function that given an arbitrary a produces a proof of $P(a)$. In Alfa this corresponds to the following definition:

$$\begin{array}{ll} \forall\ (B \in A \to \mathit{Type}) \in \mathit{Type} \qquad [A \in \mathit{Type}] \\ \forall B \equiv \mathbf{data}\,\{\, \forall_{\mathrm{I}}\ (p \in (a \in A) \to B\ a) \,\} \end{array}$$

This definition introduces the forall type together with a data constructor $\forall_{\mathrm{I}}$ to construct values of this type. Apart from this we also need to instantiate $\forall$-quantified values. This can be done using the function $\forall_{\mathrm{E}}$ defined by

$$\begin{array}{ll} \forall_{\mathrm{E}}\ (t \in A, P \in \forall B) \in B\ t \qquad [A \in \mathit{Type}, B \in A \to \mathit{Type}] \\ \forall_{\mathrm{E}}\ t\ (\forall_{\mathrm{I}}\ p) \equiv p\ t \end{array}$$

When constructing values of the type $\forall B$ we often write expressions of the form $\forall_{\mathrm{I}}(\lambda a.\, P(a))$. So often, in fact, that we introduce a little syntactic sugar for this pattern. We omit the λ and let $\forall_{\mathrm{I}}$ be a binding construct, so instead of writing $\forall_{\mathrm{I}}(\lambda a.\, P(a))$ we write $\forall_{\mathrm{I}}\, a.\, P(a)$. This notation is not supported by Alfa, but used only for this presentation.

6.2.4 Inductively defined families of sets

A recent feature added to Alfa is the ability to write inductive definitions of families of sets. Basically this means that we can define a family of datatypes indexed by some set S using one datatype definition. For instance we can define an identity relation over sets as follows.

$$\begin{array}{ll} (==)\ (a \in A) \in A \to \mathit{Set} \qquad [A \in \mathit{Set}] \\ (a\ ==) \equiv \mathbf{data}\,\{\, \mathbf{refl} \in (a == a) \,\} \end{array}$$

This definition definition states that for a set A and an object $a \in A$, $(a\ ==)$ is a family of sets index by A. So for $b \in A$, $a == b$ is the set containing proofs that a is identical to b. The definition introduces one constructor **refl** that constructs an element of the set $a == a$. Note that since **refl** is the only constructor we can only get an element of $a == b$ is a and b are the same object.

To see how to use this identity type lets look at a very short example. We can define the natural numbers as a new datatype in Alfa.

$$\begin{array}{l} Nat \in Set \\ Nat \equiv \mathbf{data} \left\{ \begin{array}{l} \mathbf{Z} \\ \mathbf{S}\ (n \in Nat) \end{array} \right\} \end{array}$$

Suppose now that we want a proof that the successor constructor respects identity, that is if n and m are identical then so are $\mathbf{S}\ n$ and $\mathbf{S}\ m$. This proof is trivial using our identity family:

$$\begin{array}{ll} \mathit{SuccPreserveId}\ (id \in n == m) \in \mathbf{S}\ n == \mathbf{S}\ m & [n, m \in Nat] \\ \mathit{SuccPreserveId}\ \mathbf{refl} \equiv \mathbf{refl} & \end{array}$$

The proof that n is identical to m must be a **refl** since that is the only way to construct identity proofs. But then n and m is really the same object and hence $\mathbf{S}\ n$ is the same as $\mathbf{S}\ m$ in which we can use **refl** as the requested proof. The nice thing with this identity relation is that we get congruence for free, and it is in some sense polymorphic, that is it works for any set A.

In the next section we are going to use this identity relation, but on the type level. So we lift the definition of (==) to operate on types instead of on sets, simply by replacing all occurrences of *Set* by *Type*:

$$\begin{array}{ll} (==)\ (a \in A) \in A \to \mathit{Type} & [A \in \mathit{Type}] \\ (a\ ==) \equiv \mathbf{data}\,\{\,\mathbf{refl} \in (a == a)\,\} & \end{array}$$

Chapter 7

Generic Haskell in type theory

7.1 Introduction

Currently Generic Haskell lacks a type system and the compiler performs only rudimentary checking of the programs to be compiled. This means that a type error in the Generic Haskell code causes the Generic Haskell compiler to generate faulty Haskell code, in which case the error is reported by the Haskell compiler instead of the Generic Haskell compiler. These error messages are quite hard to understand and relate to the Generic Haskell code which makes debugging a difficult task.

This chapter describes an encoding of Generic Haskell into type theory using the Alfa system described in section 6. Our hope is that by expressing generic programs in a dependent types framework one can get a better understanding of how to design a type system for Generic Haskell. This encoding also gives an alternative semantics to Generic Haskell program that is perhaps easier to reason about than the specialization semantics used by the Generic Haskell compiler.

Our approach is to encode a type indexed function as a function taking a datatype representation as its first argument, similarly a kind indexed type is a function from a kind representation to a type. We do not consider type indexed types here but we see no reason why there should be any problems with that.

Many of the ideas in this chapter have been borrowed from a paper by Altenkirch and McBride [AM02] discussing how to use generic programming ideas in a dependently typed setting.

A short note on terminology before we start. In this chapter we use the word datatype for all haskell types.

7.2 Kinds and signatures

First of all we are going to need to represent kinds. For this purpose we define the set *Kind* with constructors $\star$ and $(\to)$:

$$\begin{array}{lcl} \mathit{Kind} & \in & \mathit{Set} \\ \mathit{Kind} & \equiv & \textbf{data} \left\{ \begin{array}{l} \star \\ (\to)\ (k, l \in \mathit{Kind}) \end{array} \right\} \end{array}$$

Members of this set include $\star$, $\star \to \star$ and $(\star \to \star) \to \star$. We also want to be able to convert from a kind representation to a real kind. This is done by the function *kind* by recursion over the representation.

$$\begin{array}{lcl} \mathit{kind}\ (k' \in \mathit{Kind}) & \in & \mathit{Type} \\ \mathit{kind}\ \star & \equiv & \mathit{Set} \\ \mathit{kind}\ (k \to l) & \equiv & \mathit{kind}\ k \to \mathit{kind}\ l \end{array}$$

This agrees with our intuition: an object of kind $\star$ is just an ordinary set and an object of kind $k \to l$ maps objects of kind k to objects of kind l. Note that the arrow on the right hand side of the definition is the function type constructor and the arrow on the left hand side is the data constructor in the set *Kind*.

We also need signatures which are basically lists of kind representations. Signatures are captured by the set *Sig* with constructors ε and $(;)$:

$$\begin{array}{lcl} \mathit{Sig} & \in & \mathit{Set} \\ \mathit{Sig} & \equiv & \textbf{data} \left\{ \begin{array}{l} \varepsilon \\ (;)\ (\Sigma \in \mathit{Sig}, k \in \mathit{Kind}) \end{array} \right\} \end{array}$$

Here we could have used a parameterized list datatype, but we chose to define our own signature type mostly because we then get a special syntax for signatures that conforms to the syntax for environments as we shall see later on. An example of a signature is $\varepsilon; \star; \star \to \star \in \mathit{Sig}$. Note that the head of the list is the right-most element as opposed to the familiar Haskell lists where the head of the list is to the left.

7.3 Variables

We use deBruijn indices to represent variables, that is instead of named variables a variable is represented by a natural number indicating the distance from the occurrence of the variable to where it is bound. For instance the lambda expression $\lambda x.\lambda y.\lambda z.xz(yz)$ would be represented by $\lambda\lambda\lambda 2\ 0\ (1\ 0)$ using deBruijn indices instead of named variables. Since we want typed variables we cannot use ordinary natural numbers, instead we define a family of sets *Var*, indexed by signatures, so that an object in the set $\mathit{Var}\ k\ \Sigma$ represents a variable of kind k, where the signature Σ is the current kind environment, containing the kinds of all the bound variables.

$$\begin{array}{l} \mathit{Var}\ (k \in \mathit{Kind}) \in \mathit{Sig} \to \mathit{Set} \\ \mathit{Var}\ k \equiv \mathbf{data} \left\{ \begin{array}{l} \mathbf{vz}\ (\Sigma \in \mathit{Sig}) \in \mathit{Var}\ k\ (\Sigma; k) \\ \mathbf{vs}\ (\Sigma \in \mathit{Sig}, l \in \mathit{Kind}, v \in \mathit{Var}\ k\ \Sigma) \in \mathit{Var}\ k\ (\Sigma; l) \end{array} \right\} \end{array}$$

The only way to form a variable is by using one of the constructors **vz** and **vs** corresponding to the natural number zero and the successor function respectively. If Σ is an arbitrary signature then $\mathbf{vz}\ \Sigma \in \mathit{Var}\ k\ (\Sigma; k)$ for any kind k. That is the kind of the variable **vz** is the same as the kind of the closest bound variable, which is good since **vz** is supposed to represent this variable. We omit the parameter Σ to **vz** to improve readability whenever it can be inferred from the context. The definition of the constructor **vs** states that if Σ is a signature, l is a kind and v is a variable of kind k in the environment Σ, then $\mathbf{vs}\ \Sigma\ l\ v \in \mathit{Var}\ k\ (\Sigma; l)$. In other words if we have a representation of a variable of kind k in the environment Σ we can get a representation of this variable in the extended environment $\Sigma; l$ by using the constructor **vs**. We omit the arguments Σ and l in most cases. Using this argument hiding the variables look just like ordinary natural numbers with explicit use of succ and zero. For instance, **vs** (**vs vz**) corresponds to the natural number 2. Note that because both **vz** and **vs** yields variables over non-empty signatures, the set $\mathit{Var}\ k\ \varepsilon$ is empty for all kinds k, corresponding to the fact that we can only use variables that have been bound by a previous abstraction.

7.4 Environments

An environment is basically a mapping from variables to data, where the data can be for instance the expression bound to a variable during evaluation. A common way of implementing environments is as a list of variable data pairs, but since we are not using named variables we do not need to keep the variables in the list. An ordinary list is not sufficient for our purposes, however, since we want to be able to store data that depends on the kind of the variable it is bound to. For example we want to associate only types of kind k to variables of kind k. To be able to do this we parameterize the environment by a signature.

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathit{Env}\ (F \in \mathit{Kind} \to \mathit{Type}, \Sigma' \in \mathit{Sig}) \in \mathit{Type}} \\ \mathit{Env}\ F\ \varepsilon & \equiv & \mathbf{data}\ \{\varepsilon\} \\ \mathit{Env}\ F\ (\Sigma; k) & \equiv & \mathbf{data}\ \{(;)\ (\Gamma \in \mathit{Env}\ F\ \Sigma, f \in F\ k)\} \end{array}$$

Note that we choose to store types rather than sets in the environment. An environment over the empty signature is just an empty environment ε (we use the same notation for both environments and signatures but hopefully this should not lead to any confusion) and an environment over the signature $\Sigma; k$ is an environment Γ over the signature Σ followed by an object f from the type $F\,k$.

To look up a value in an environment we just have to pick out the value corresponding the variable we are interested in.

$$\begin{array}{lll}
\multicolumn{3}{l}{(.)\ (\Gamma \in \mathit{Env}\ F\ \Sigma, v \in \mathit{Var}\ k\ \Sigma) \in F\,k} \\
\multicolumn{3}{l}{\quad [F \in \mathit{Kind} \to \mathit{Type}, \Sigma \in \mathit{Sig}, k \in \mathit{Kind}]} \\
(\Gamma'; f).\mathbf{vz} & \equiv & f \\
(\Gamma'; f).(\mathbf{vs}\ v') & \equiv & \Gamma'.v'
\end{array}$$

The lookup function is defined by pattern matching on the variable v. If v is $\mathbf{vz}$ then Σ is of the form $\Sigma'; k$ and thus Γ must be of the form $\Gamma'; f$ where $f \in F\,k$. Otherwise v is of the form $\mathbf{vs}\ v$, in which case Σ must be $\Sigma'; l$ for some Σ' and l. We can then throw away the head of the environment and lookup v in the rest recursively. Note that we do not need a case for the empty environment since if Γ is empty then Σ is the empty signature and there is no variable v in the set $Var\ k\ \varepsilon$.

Besides looking up values we want to be able to apply a function to all values in an environment. What we want is the standard list mapping function extended to our dependent lists. This turns out to be as straight-forward as we would expect.

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{mapEnv}\ (\Sigma \in \mathit{Sig}, \eta \in (k \in \mathit{Kind}) \to F\,k \to G\,k, \Gamma \in \mathit{Env}\ F\ \Sigma)} \\
\multicolumn{3}{l}{\quad \in \mathit{Env}\ G\ \Sigma \qquad [F, G \in \mathit{Kind} \to \mathit{Type}]} \\
\mathit{mapEnv}\ \varepsilon\ \eta\ \varepsilon & \equiv & \varepsilon \\
\mathit{mapEnv}\ (\Sigma'; k)\ \eta\ (\Gamma'; f) & \equiv & \mathit{mapEnv}\ \Sigma'\ \eta\ \Gamma'; \eta\ k\ f
\end{array}$$

Since the values in the environment are indexed by kinds the function η that we want to map over the environment takes a kind index k as its first argument. When we apply η in the definition we have to supply the kind as well as the value from the environment.

As soon as we have functions we get laws that the functions have to abide by. It turns out that the law that we need is the one stating that mapping a function over an environment and then looking up a value in the resulting environment yields the same result as looking up the value in the original environment and then applying the function. Formalized in Alfa this becomes a proof constructing function:

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathit{mapLookEnvCommute} \left(\begin{array}{l} \Gamma \in \mathit{Env}\ F\ \Sigma, \\ \eta \in (k \in \mathit{Kind}) \to F\,k \to G\,k, \\ k \in \mathit{Kind}, v \in \mathit{Var}\ k\ \Sigma \end{array} \right)} \\
\multicolumn{3}{l}{\quad \in \eta\ k\ (\Gamma.v) == (\mathit{mapEnv}\ \eta\ \Gamma).v} \\
\multicolumn{3}{l}{\quad [\Sigma \in \mathit{Sig}, F, G \in \mathit{Kind} \to \mathit{Type}]} \\
\mathit{mapLookEnvCommute}\ (\Gamma'; f)\ \eta\ k\ \mathbf{vz} & \equiv & \mathbf{refl} \\
\mathit{mapLookEnvCommute}\ (\Gamma'; f)\ \eta\ k\ (\mathbf{vs}\ v') & \equiv & \\
\multicolumn{3}{l}{\quad \mathit{mapLookEnvCommute}\ \Gamma'\ \eta\ k\ v'}
\end{array}$$

$$
\begin{array}{l}
\mathit{OpenType}\ (\Sigma \in \mathit{Sig}) \in \mathit{Kind} \to \mathit{Set} \\
\mathit{OpenType}\ \Sigma \equiv \\
\quad \textbf{data} \left\{ \begin{array}{l}
\textbf{Sum} \in \mathit{OpenType}\ \Sigma\ (\star \to \star \to \star) \\
\textbf{Prod} \in \mathit{OpenType}\ \Sigma\ (\star \to \star \to \star) \\
\textbf{1} \in \mathit{OpenType}\ \Sigma\ \star \\
\textbf{Con}\ (c \in \mathit{ConDescr}) \in \mathit{OpenType}\ \Sigma\ (\star \to \star) \\
\textbf{Label}\ (l \in \mathit{LabelDescr}) \in \mathit{OpenType}\ \Sigma\ (\star \to \star) \\
(\to) \in \mathit{OpenType}\ \Sigma\ (\star \to \star \to \star) \\
\textbf{Int}\ \in \mathit{OpenType}\ \Sigma\ \star \\
\lambda \left(\begin{array}{l} k, l \in \mathit{Kind}, \\ e \in \mathit{OpenType}\ (\Sigma; k)\ l \end{array} \right) \in \mathit{OpenType}\ \Sigma\ (k \to l) \\
(\diamond) \left(\begin{array}{l} k, l \in \mathit{Kind}, \\ f \in \mathit{OpenType}\ \Sigma\ (k \to l), \\ e \in \mathit{OpenType}\ \Sigma\ k \end{array} \right) \in \mathit{OpenType}\ \Sigma\ l \\
\textbf{V}\ (k \in \mathit{Kind}, v \in \mathit{Var}\ k\ \Sigma) \in \mathit{OpenType}\ \Sigma\ k
\end{array} \right\}
\end{array}
$$

Figure 7.1: Datatype representations

The proof is by induction over the variable. If the variable is **vz** then the equality follows directly from the definitions of (.) and *mapEnv*. Remember that **refl** is the single member of the type $a == a$ for an object a. If the variable is of the form **vs** v then it is enough to prove that looking up v in the tail Γ' of Γ and then applying η is the same as mapping η over Γ' and then looking up v. Since this is the induction hypothesis we are done.

7.5 Datatype representations

Now we finally have enough machinery to be able to define the representation of a Haskell datatype. In figure 7.1 we define a family of sets *OpenType* Σ indexed by kinds. An object in the set *OpenType* Σ k represents a datatype of kind k where Σ is the signature containing the kinds of the available variables. So for any Σ, **Sum** and **Prod** are representations of datatypes of kind $\star \to \star \to \star$ and **1** represents a datatype of kind $\star$. The actual datatypes represented by **Sum**, **Prod** and **1** are (:+:), (:*:) and Unit that we encountered in chapter 4. Notice that the constructors **Con** and **Label** contain a constructor and label descriptor resepectively. This agrees with the fact that you get hold of this descriptor when you define a case for Con or Label in a type indexed function in Generic Haskell. **Int** and $(\to)$ represents the obvious datatypes. One should have similar representations for all primitive types but we do not want to clutter up the code too much. The three last constructors are the most interesting ones.

If e represents a datatype of kind l with free variables of kinds $\Sigma; k$ then $\lambda\ k\ l\ e$ represents a datatype of kind $k \to l$ with free variables of kinds Σ. So λ abstracts e over the variable **vz**. Usually we omit the kinds k and l and just write λe.

The constructor $(\diamond)$ represents the application of one datatype to another so if we have f representing a datatype of kind $k \to l$ and e representing a datatype of kind k, then $(\diamond)\ k\ l\ f\ e$ represents application of these datatypes with the resulting kind l. As with the constructor λ we often omit the kind arguments and write $(\diamond)$ as infix, so we get the more readable form $f \diamond e$.

The final constructor $\mathbf{V}$ represents a variable occurrence. If v is a variable of kind k then $\mathbf{V}\ k\ v$ represents the datatype of kind k consisting of only the variable v. We omit the kind argument and write $\mathbf{V}\,v$ most of the time.

Before we give any examples of datatype representations we define two small functions to improve the readability of our representation types.

$$(+)\ (t, t' \in \mathit{OpenType}\ \Sigma\ \star) \in \mathit{OpenType}\ \Sigma\ \star \qquad [\Sigma \in \mathit{Sig}]$$
$$t + t' \equiv \mathbf{Sum} \diamond t \diamond t'$$

$$(*)\ (t, t' \in \mathit{OpenType}\ \Sigma\ \star) \in \mathit{OpenType}\ \Sigma\ \star \qquad [\Sigma \in \mathit{Sig}]$$
$$t * t' \equiv \mathbf{Prod} \diamond t \diamond t'$$

Now we can define representations for our favorite datatypes:

$$\mathsf{Nat}^\circ \in \mathit{OpenType}\ \Sigma\ \star \qquad [\Sigma \in \mathit{Sig}]$$
$$\mathsf{Nat}^\circ \equiv \mathbf{Con}\ c_{\mathsf{Zero}} \diamond \mathbf{1} + \mathbf{Con}\ c_{\mathsf{Succ}} \diamond \mathsf{Nat}^\circ$$

$$\mathsf{List}^\circ \in \mathit{OpenType}\ \Sigma\ (\star \to \star) \qquad [\Sigma \in \mathit{Sig}]$$
$$\mathsf{List}^\circ \equiv \lambda(\mathbf{Con}\ c_{[]} \diamond \mathbf{1} + \mathbf{Con}\ c_{(:)} \diamond (\mathbf{V\,vz} * \mathsf{List}^\circ \diamond \mathbf{V\,vz}))$$

$$\mathsf{Fix}^\circ \in \mathit{OpenType}\ \Sigma\ ((\star \to \star) \to \star) \qquad [\Sigma \in \mathit{Sig}]$$
$$\mathsf{Fix}^\circ \equiv \lambda(\mathbf{Con}\ c_{\mathsf{In}} \diamond (\mathbf{Label}\ l_{out} \diamond (\mathbf{V\,vz} \diamond (\mathsf{Fix}^\circ \diamond \mathbf{V\,vz}))))$$

We assume that we have an appropriate constructor descriptor c_D for every datatype constructor D, and a label descriptor l_L for every label L.

One difference between the structure types used in Generic Haskell and these datatype representations is that the structure types used the real datatype in the recursive call. For instance we had

$$\textbf{type}\ \mathsf{List}^\circ\ a = \mathsf{Con\ Unit}\ {:+:}\ \mathsf{Con}\ (a\ {:*:}\ [a])$$

whereas the dependent type representation of the list datatype calls itself recursively. We could not use the real list type in the recursive call since the datatype representation is an ordinary value and not a datatype.

7.6 Converting to real types

The next step is to formalize which datatype is actually represented by a specific representation. First of all we need to define the structure types we want to

$$(\mathrel{:+:})\ (A, B \in Set) \in Set$$
$$A \mathrel{:+:} B \equiv \textbf{data} \left\{ \begin{array}{l} \mathsf{Inl}\ (a \in A) \\ \mathsf{Inr}\ (b \in B) \end{array} \right\}$$

$$(\mathrel{:*:})\ (A, B \in Set) \in Set$$
$$A \mathrel{:*:} B \equiv \textbf{data}\ \{(\mathrel{:*:})\ (a \in A, b \in B)\}$$

$$\mathsf{Unit} \in Set$$
$$\mathsf{Unit} \equiv \textbf{data}\ \{\mathsf{Unit}\}$$

$$\mathsf{Con}\ (A \in Set) \in Set$$
$$\mathsf{Con}\ A \equiv \textbf{data}\ \{\mathsf{Con}\ (a \in A)\}$$

$$\mathsf{Label}\ (A \in Set) \in Set$$
$$\mathsf{Label}\ A \equiv \textbf{data}\ \{\mathsf{Label}\ (a \in A)\}$$

$$\mathsf{Fun}\ (A, B \in Set) \in Set$$
$$\mathsf{Fun}\ A\ B \equiv A \to B$$

$$\mathsf{Int} \in Set$$
$$\mathsf{Int} \equiv \ldots$$

Figure 7.2: The structure types in Alfa

build our datatypes from. The Alfa equivalents to the structure types defined in figure 4.1 are shown in figure 7.2. The actual definition of the Int datatype is not relevant to our purposes and is thus excluded.

Now given a datatype representation $t \in OpenType\ \Sigma\ k$ we want to construct the datatype that t represents. To do this we need an environment mapping the free variables in t to datatypes of the appropriate kind. The function *openType* defined in figure 7.3 constructs this datatype in the natural way. One thing to note is that in the λ-case the λ in the left hand side is the constructor of *OpenType* and the one in the right hand side is a "real" λ. Note also that the constructor and label descriptors only exist in the datatype representations and not in the actual datatypes.

When we are dealing with recursive types both the datatype representation and the datatype are infinite structures. This is quite natural because a recursive datatype indeed is infinite. The problem is that type theory does not acknowledge infinite structures as something natural. This means that if we want this embedding to be a semantics for Generic Haskell we have to find a way around the infinite structures, but as long as we only use the embedding to experiment with Generic Haskell we can consider these problems to be a flaw in type theory rather than in our embedding.

$$
\begin{array}{lll}
\multicolumn{3}{l}{openType\ (\Phi \in Env\ kind\ \Sigma, t \in OpenType\ \Sigma\ k) \in kind\ k} \\
\multicolumn{3}{l}{\quad [\Sigma \in Sig, k \in Kind]} \\
openType\ \Phi\ \mathbf{Sum} & \equiv & (:+:) \\
openType\ \Phi\ \mathbf{Prod} & \equiv & (:*:) \\
openType\ \Phi\ \mathbf{1} & \equiv & \mathsf{Unit} \\
openType\ \Phi\ (\mathbf{Con}\ c) & \equiv & \mathsf{Con} \\
openType\ \Phi\ (\mathbf{Label}\ l) & \equiv & \mathsf{Label} \\
openType\ \Phi\ (\rightarrow) & \equiv & \mathsf{Fun} \\
openType\ \Phi\ \mathbf{Int} & \equiv & \mathsf{Int} \\
openType\ \Phi\ (\lambda e) & \equiv & \lambda e'.\, openType\ (\Phi; e')\ e \\
openType\ \Phi\ (f \diamond e) & \equiv & (openType\ \Phi\ f)\ (openType\ \Phi\ e) \\
openType\ \Phi\ (\mathbf{V}\, v) & \equiv & \Phi.v
\end{array}
$$

Figure 7.3: Converting a representation to a datatype

7.7 Implementing the generic map function

The framework we have developed in this chapter is sufficient to express most type indexed functions. In this section we show how to define the generic map function *gmap* from chapter 4 in this framework.

The kind indexed type *Map* is defined almost exactly as in Generic Haskell.

$$
\begin{array}{lll}
\multicolumn{3}{l}{Map\ (k \in Kind, t_1, t_2 \in kind\ k) \in Type} \\
Map\ \star\ t_1\ t_2 & \equiv & t_1 \rightarrow t_2 \\
Map\ (k \rightarrow l)\ t_1\ t_2 & \equiv & \forall u_1.\forall u_2.Map\ k\ u_1\ u_2 \rightarrow Map\ l\ (t_1\ u_1)\ (t_2\ u_2)
\end{array}
$$

The only difference being that it is typed and that there are no funny brackets around the kind argument.

The definition of the type indexed function *gmap* is a little more involved. We start by giving the type that we would like *gmap* to have.

$$
\begin{array}{l}
gmap\ (t \in OpenType\ \varepsilon\ k) \\
\quad \in Map\ k\ (openType\ \varepsilon\ t)\ (openType\ \varepsilon\ t) \quad [k \in Kind]
\end{array}
$$

This is a direct translation of the type given to *gmap* in Generic Haskell. The only difference is that now we must be explicit about what is a datatype representation and what is a real datatype. So the argument t to *gmap* is a representation of a datatype with no free variables, and $openType\ \varepsilon\ t$ is the datatype t represents. Naturally we only want datatypes with no free variables as arguments to *gmap*. The problem is that we want to recurse over the datatype representation and since t might contain subexpressions with free variables we have to be able to handle this. The natural way of doing this is by having *gmap* take an environment containing bindings for all the free variables, as an extra parameter. Since we want to keep the nice type of *gmap* given above, we are going to call the more general mapping function *gmap′* and define *gmap* is terms of *gmap′*.

$$gmap\ t \equiv gmap'\ \varepsilon\ t$$

Remember that t does not contain any free variables so we can pass the empty environment to *gmap′*. Now the tricky question is to decide what we need to bind to the variables. To answer this we first look at a small example. Consider the identity type whose type representation is

$$\begin{aligned}&\mathsf{Id}^{\circ} \in \mathit{OpenType}\ \varepsilon\ (\star \to \star)\\&\mathsf{Id}^{\circ} \equiv \lambda(\mathbf{V}\ \mathbf{vz})\end{aligned}$$

We can check that Id° really represents the identity type by performing the following calculation.

$$\begin{aligned}\mathit{openType}\ \varepsilon\ \mathsf{Id}^{\circ} &\equiv \mathit{openType}\ \varepsilon\ (\lambda(\mathbf{V}\ \mathbf{vz}))\\&\equiv \lambda e.\, \mathit{openType}\ (\varepsilon; e)\ (\mathbf{V}\ \mathbf{vz})\\&\equiv \lambda e.\, (\varepsilon; e).\mathbf{vz}\\&\equiv \lambda e.\, e\end{aligned}$$

If we apply *gmap* to Id° we should get something of the type

$$\begin{aligned}\mathit{gmap}\ \mathsf{Id}^{\circ} &\in \mathit{Map}\ (\star \to \star)\ (\lambda e.e)\ (\lambda e.e)\\&\equiv \forall u_1 \forall u_2 . \mathit{Map}\ \star\ u_1\ u_2 \to \mathit{Map}\ \star\ ((\lambda e.e)\ u_1)\ ((\lambda e.e)\ u_2)\\&\equiv \forall u_1 \forall u_2 . (u_1 \to u_2) \to (u_1 \to u_2)\end{aligned}$$

Since the outer-most constructor of Id° is a λ, lets look at what *gmap′* should do in that case. The datatype representation λe has type $\mathit{OpenType}\ \Sigma\ (k \to l)$ for some Σ, k and l, where $e \in \mathit{OpenType}\ (\Sigma; k)\ l$. In our case $\Sigma = \varepsilon$ and $k = l = \star$. What we want is that the λ-case should take care of the part "$\forall u_1 \forall u_2 . (u_1 \to u_2) \to$" and call *gmap′* recursively on $\mathbf{V}\,\mathbf{vz}$ to get the final $u_1 \to u_2$. To do this we can define the following

$$\mathit{gmap}'\ \Phi\ (\lambda e) \equiv \forall_{\mathrm{I}} u_1 u_2 .\, \lambda m_U .\, \mathit{gmap}'\ (\Phi; ?)\ e$$

We still haven't specified the type of *gmap′*, but this seems like a reasonable definition of the λ-case, provided we find something clever to replace the question mark by. There is a lot of hidden information in the definition above, for instance the kinds k and l and the types $u_1, u_2 \in \mathit{kind}\ k$ and $m_U \in \mathit{Map}\ k\ u_1\ u_2$. Now in our case we want

$$\mathit{gmap}'\ (\Phi; ?)\ (\mathbf{V}\ \mathbf{vz}) \equiv m_U \in u_1 \to u_2$$

From this we can see that the questionmark should contain u_1, u_2 and m_U, for instance we could take it to be the triple $u_1 \times u_2 \times m_U$. In general we define a *Mapping* as follows:

$$\begin{array}{l} \mathit{Mapping}\ (k \in \mathit{Kind}) \in \mathit{Type} \\ \mathit{Mapping}\ k \equiv \mathit{kind}\ k \times (\lambda u_1 .\, \mathit{kind}\ k \times (\lambda u_2 .\, \mathit{Map}\ k\ u_1\ u_2)) \end{array}$$

We also define three extraction functions *first*, *second* and *third* in the obvious way.

$$\begin{array}{l} \mathit{first}\ (k \in \mathit{Kind}, m \in \mathit{Mapping}\ k) \in \mathit{kind}\ k \\ \mathit{first}\ k\ (u_1 \times \_ \times \_) \equiv u_1 \\ \\ \mathit{second}\ (k \in \mathit{Kind}, m \in \mathit{Mapping}\ k) \in \mathit{kind}\ k \\ \mathit{second}\ k\ (\_ \times u_2 \times \_) \equiv u_2 \\ \\ \mathit{third}\ (k \in \mathit{Kind}, m \in \mathit{Mapping}\ k) \in \mathit{Map}\ k\ (\mathit{first}\ k\ m)\ (\mathit{second}\ k\ m) \\ \mathit{third}\ k\ (\_ \times \_ \times m_U) \equiv m_U \end{array}$$

Now we are finally ready to give the type of *gmap′*:

$$\begin{array}{l} \mathit{gmap}'\ (\Phi \in \mathit{Env}\ \mathit{Mapping}\ \Sigma, t \in \mathit{OpenType}\ \Sigma\ k) \\ \quad \in \mathit{Map}\ k \quad (\mathit{openType}\ (\mathit{mapEnv}\ \mathit{first}\ \Phi)\ t) \\ \qquad\qquad\quad (\mathit{openType}\ (\mathit{mapEnv}\ \mathit{second}\ \Phi)\ t) \\ \quad [\Sigma \in \mathit{Sig}, k \in \mathit{Kind}] \end{array}$$

So if t is datatype representation of kind k and Φ is an environment associating the free variables in t with mappings, *gmap′* Φ t is a map from t interpreted in the environment containing all the first components of Φ to t interpreted in the environment containing all the second components. For instance we get

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{gmap}'\ (\varepsilon; u_1 \times u_2 \times m_U)\ (\mathbf{V}\, \mathbf{vz})} \\ \in & \mathit{Map}\ \star\ \ (\mathit{openType}\ (\mathit{mapEnv}\ \mathit{first}\ (\varepsilon; u_1 \times u_2 \times m_U))\ (\mathbf{V}\, \mathbf{vz})) \\ & \qquad\quad (\mathit{openType}\ (\mathit{mapEnv}\ \mathit{second}\ (\varepsilon; u_1 \times u_2 \times m_U))\ (\mathbf{V}\, \mathbf{vz})) \\ \equiv & \mathit{Map}\ \star\ (\mathit{openType}\ (\varepsilon; u_1)\ (\mathbf{V}\, \mathbf{vz}))\ (\mathit{openType}\ (\varepsilon; u_2)\ (\mathbf{V}\, \mathbf{vz})) \\ \equiv & \mathit{Map}\ \star\ u_1\ u_2 \\ \equiv & u_1 \to u_2 \end{array}$$

Which is exactly what we want. The definition of *gmap′* follows the Generic Haskell definition closely, the difference is that now we have to be explicit about the forall quantifiers.

$$\begin{array}{lcl}
\mathit{gmap}'\ \Phi\ \mathbf{Sum} & \equiv & \forall_{\mathrm{I}} A_1 A_2.\, \lambda m_A.\, \forall_{\mathrm{I}} B_1 B_2.\, \lambda m_B. \\
 & & \lambda e.\ \ \mathbf{case}\ e\ \mathbf{of} \\
 & & \qquad \mathsf{Inl}\ x \quad \rightarrow \quad \mathsf{Inl}\ (m_A\ x) \\
 & & \qquad \mathsf{Inr}\ y \quad \rightarrow \quad \mathsf{Inr}\ (m_B\ y) \\
\mathit{gmap}'\ \Phi\ \mathbf{Prod} & \equiv & \forall_{\mathrm{I}} A_1 A_2.\, \lambda m_A.\, \forall_{\mathrm{I}} B_1 B_2.\, \lambda m_B. \\
 & & \lambda(x :\!*\!: y).\, m_A\ x :\!*\!: m_B\ y \\
\mathit{gmap}'\ \Phi\ \mathbf{1} & \equiv & \mathit{id} \\
\mathit{gmap}'\ \Phi\ (\mathbf{Con}\ c) & \equiv & \forall_{\mathrm{I}} A_1 A_2.\, \lambda m_A.\, \lambda(\mathsf{Con}\ x).\, \mathsf{Con}\ (m_A\ x) \\
\mathit{gmap}'\ \Phi\ (\mathbf{Label}\ l) & \equiv & \forall_{\mathrm{I}} A_1 A_2.\, \lambda m_A.\, \lambda(\mathsf{Label}\ x).\, \mathsf{Label}\ (m_A\ x) \\
\mathit{gmap}'\ \Phi\ (\rightarrow) & \equiv & \mathbf{error}\ \text{“Undefined”} \\
\mathit{gmap}'\ \Phi\ \mathbf{Int} & \equiv & \mathit{id}
\end{array}$$

Of the remaining cases we have already seen the λ-case. In the application case we have to explicitly eliminate the *forall*s, otherwise we just apply $\mathit{gmap}'\ \Phi\ f$ to $\mathit{gmap}'\ \Phi\ e$.

$$\begin{array}{lcl}
\mathit{gmap}'\ \Phi\ (\lambda e) & \equiv & \forall_{\mathrm{I}} u_1 u_2.\, \lambda m_U.\, \mathit{gmap}'\ (\Phi; u_1 \times u_2 \times m_U)\ e \\
\mathit{gmap}'\ \Phi\ (f \diamond e) & \equiv & \forall_{\mathrm{E}}\ (\mathit{openType}\ (\mathit{mapEnv}\ \mathit{second}\ \Phi)\ e) \\
 & & \left(\begin{array}{ll} \forall_{\mathrm{E}} & (\mathit{openType}\ (\mathit{mapEnv}\ \mathit{first}\ \Phi)\ e) \\ & (\mathit{gmap}'\ \Phi\ f) \end{array} \right) \\
 & & (\mathit{gmap}'\ \Phi\ e)
\end{array}$$

The last case is the variable case. What we would like to do in this case is just to return the mapping function bound to the variable. We could try to define

$$\mathit{gmap}'\ \Phi\ (\mathbf{V}\ v) \quad \equiv \quad \mathit{third}\ k\ (\Phi.v)$$

However, this expression does not have the correct type.

$$\begin{array}{lcl}
\mathit{gmap}'\ \Phi\ (\mathbf{V}\ v) & \in & \mathit{Map}\ k\ \ (\mathit{openType}\ (\mathit{mapEnv}\ \mathit{first}\ \Phi)\ (\mathbf{V}\ v)) \\
 & & \qquad\qquad (\mathit{openType}\ (\mathit{mapEnv}\ \mathit{second}\ \Phi)\ (\mathbf{V}\ v)) \\
 & \equiv & \mathit{Map}\ k\ \ ((\mathit{mapEnv}\ \mathit{first}\ \Phi).v) \\
 & & \qquad\qquad ((\mathit{mapEnv}\ \mathit{second}\ \Phi).v) \\
 & & \\
\mathit{third}\ k\ (\Phi.v) & \in & \mathit{Map}\ k\ (\mathit{first}\ k\ (\Phi.v))\ (\mathit{second}\ k\ (\Phi.v))
\end{array}$$

Fortunately we have already proven (in section 7.4) that mapping a function over an environment and then performing a lookup yields the same result as first performing the lookup and then applying the function, so using substitution together with instances of *mapLookEnvCommute* we can make $\mathit{third}\ k\ (\Phi.v)$

have the right type. We do this using the function *cast* defined by

$$\begin{array}{lll}
cast\ (m \in Map\ k\ (first\ k\ (\Phi.v))\ (second\ k\ (\Phi.v))) & & \\
 & \in & Map\ k\ ((mapEnv\ first\ \Phi).v)\ ((mapEnv\ second\ \Phi).v) \\
 & & [\Sigma \in Sig, \Phi \in Env\ Mapping\ \Sigma, k \in Kind, v \in Var\ k\ \Sigma] \\
cast\ m & \equiv & subst\ (\lambda t_1.\, Map\ k\ t_1\ ((mapEnv\ second\ \Phi).v)) \\
 & & \quad (mapLookEnvCommute\ \Phi\ first\ k\ v) \\
 & & \quad \left(\begin{array}{l} subst \\ \quad (\lambda t_2.\, Map\ k\ (first\ k\ (\Phi.v))\ t_2) \\ \quad (mapLookEnvCommute\ \Phi\ second\ k\ v) \\ \quad m \end{array} \right)
\end{array}$$

The final version of the variable case of *gmap′* can then be written as

$$gmap'\ \Phi\ (\mathbf{V}\ v) \equiv cast\ (third\ k\ (\Phi.v))$$

This was the last case in the definition of *gmap′*. The complete definition can be found in figure 7.4.

In general the environment Φ passed to *gmap′* should contain tuples of types bound to all the generic arguments of the kind indexed type and an element of the kind indexed type applied to these arguments. Note that arguments to the kind indexed type that are not generic, i.e. arguments whose kinds are fixed, are not included in the tuples. So in the case of the generic *lreduce* function with the kind indexed type

$$\begin{array}{lll}
LReduce\ (k \in Kind, t \in kind\ k, b \in kind\ \star) \in Type & & \\
LReduce\ \star\ t\ b & \equiv & b \to t \to b \\
LReduce\ (k \to l)\ t\ b & \equiv & \forall u. LReduce\ k\ u\ b \to LReduce\ l\ (t\ u)\ b
\end{array}$$

we would get tuples of the following type in the environment:

$$\begin{array}{l}
LReduction\ (b \in kind\ \star, k \in Kind) \in Type \\
LReduction\ b\ k \equiv kind\ k \times (\lambda t.\, LReduce\ k\ t\ b)
\end{array}$$

In other words, the type b is the same in all recursive calls to *lreduce′*, which is what we would expect since b is the result type.

7.8 Conclusions

In this chapter we have described an encoding of Generic Haskell programs into type theory. We distinguish between real datatypes and their representations, encoding a type indexed function as function from a datatype representation to a function over the corresponding datatype. Likewise a kind indexed type is

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{Map}\ (k \in \mathit{Kind}, t_1, t_2 \in \mathit{kind}\ k) \in \mathit{Type}} \\
\mathit{Map}\ \star\ t_1\ t_2 & \equiv & t_1 \to t_2 \\
\mathit{Map}\ (k \to l)\ t_1\ t_2 & \equiv & \forall u_1.\forall u_2.\mathit{Map}\ k\ u_1\ u_2 \to \mathit{Map}\ l\ (t_1\ u_1)\ (t_2\ u_2)
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{Mapping}\ (k \in \mathit{Kind}) \in \mathit{Type}} \\
\mathit{Mapping}\ k & \equiv & \mathit{kind}\ k \times (\lambda u_1.\, \mathit{kind}\ k \times (\lambda u_2.\, \mathit{Map}\ k\ u_1\ u_2))
\end{array}
$$

$$
\begin{array}{lll}
\mathit{first}\ k\ m & \equiv & \mathit{fst}\ m \\
\mathit{second}\ k\ m & \equiv & \mathit{fst}\ (\mathit{snd}\ m) \\
\mathit{third}\ k\ m & \equiv & \mathit{snd}\ (\mathit{snd}\ m)
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{cast}\ (m \in \mathit{Map}\ k\ (\mathit{first}\ k\ (\Phi.v))\ (\mathit{second}\ k\ (\Phi.v)))} \\
 & \in & \mathit{Map}\ k\ ((\mathit{mapEnv}\ \mathit{first}\ \Phi).v)\ ((\mathit{mapEnv}\ \mathit{second}\ \Phi).v) \\
 & & [\Sigma \in \mathit{Sig}, \Phi \in \mathit{Env}\ \mathit{Mapping}\ \Sigma, k \in \mathit{Kind}, v \in \mathit{Var}\ k\ \Sigma] \\
\mathit{cast}\ m & \equiv & \mathit{subst}\ \ (\lambda t_1.\, \mathit{Map}\ k\ t_1\ ((\mathit{mapEnv}\ \mathit{second}\ \Phi).v)) \\
 & & \qquad (\mathit{mapLookEnvCommute}\ \Phi\ \mathit{first}\ k\ v) \\
 & & \qquad \left(\begin{array}{l} \mathit{subst} \\ \quad (\lambda t_2.\, \mathit{Map}\ k\ (\mathit{first}\ k\ (\Phi.v))\ t_2) \\ \quad (\mathit{mapLookEnvCommute}\ \Phi\ \mathit{second}\ k\ v) \\ \quad m \end{array} \right)
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{gmap}'\ (\Phi \in \mathit{Env}\ \mathit{Mapping}\ \Sigma, t \in \mathit{OpenType}\ \Sigma\ k)} \\
 & \in & \mathit{Map}\ k\ \ (\mathit{openType}\ (\mathit{mapEnv}\ \mathit{first}\ \Phi)\ t) \\
 & & \qquad\qquad (\mathit{openType}\ (\mathit{mapEnv}\ \mathit{second}\ \Phi)\ t) \\
 & & [\Sigma \in \mathit{Sig}, k \in \mathit{Kind}] \\
\mathit{gmap}'\ \Phi\ \mathbf{Sum} & \equiv & \forall_{\mathrm{I}} A_1 A_2.\, \lambda m_A.\, \forall_{\mathrm{I}} B_1 B_2.\, \lambda m_B. \\
 & & \lambda e.\ \ \mathbf{case}\ e\ \mathbf{of} \\
 & & \qquad \mathsf{Inl}\ x \quad \to \quad \mathsf{Inl}\ (m_A\ x) \\
 & & \qquad \mathsf{Inr}\ y \quad \to \quad \mathsf{Inr}\ (m_B\ y) \\
\mathit{gmap}'\ \Phi\ \mathbf{Prod} & \equiv & \forall_{\mathrm{I}} A_1 A_2.\, \lambda m_A.\, \forall_{\mathrm{I}} B_1 B_2.\, \lambda m_B. \\
 & & \lambda(x :\!*\!: y).\, m_A\ x :\!*\!: m_B\ y \\
\mathit{gmap}'\ \Phi\ \mathbf{1} & \equiv & \mathit{id} \\
\mathit{gmap}'\ \Phi\ (\mathbf{Con}\ c) & \equiv & \forall_{\mathrm{I}} A_1 A_2.\, \lambda m_A.\, \lambda(\mathsf{Con}\ x).\, \mathsf{Con}\ (m_A\ x) \\
\mathit{gmap}'\ \Phi\ (\mathbf{Label}\ l) & \equiv & \forall_{\mathrm{I}} A_1 A_2.\, \lambda m_A.\, \lambda(\mathsf{Label}\ x).\, \mathsf{Label}\ (m_A\ x) \\
\mathit{gmap}'\ \Phi\ \mathbf{Int} & \equiv & \mathit{id} \\
\mathit{gmap}'\ \Phi\ (\lambda e) & \equiv & \forall_{\mathrm{I}} u_1 u_2.\, \lambda m_U.\, \mathit{gmap}'\ (\Phi; u_1 \times u_2 \times m_U)\ e \\
\mathit{gmap}'\ \Phi\ (f \diamond e) & \equiv & \forall_{\mathrm{E}}\ \ (\mathit{openType}\ (\mathit{mapEnv}\ \mathit{second}\ \Phi)\ e) \\
 & & \qquad \left(\begin{array}{ll} \forall_{\mathrm{E}} & (\mathit{openType}\ (\mathit{mapEnv}\ \mathit{first}\ \Phi)\ e) \\ & (\mathit{gmap}'\ \Phi\ f) \end{array} \right) \\
 & & \qquad (\mathit{gmap}'\ \Phi\ e) \\
\mathit{gmap}'\ \Phi\ (\mathbf{V}\, v) & \equiv & \mathit{cast}\ (\mathit{third}\ k\ (\Phi.v))
\end{array}
$$

$$
\begin{array}{lll}
\mathit{gmap}\ (t \in \mathit{OpenType}\ \varepsilon\ k) & \in & \mathit{Map}\ k\ (\mathit{openType}\ \varepsilon\ t)\ (\mathit{openType}\ \varepsilon\ t) \\
 & & [k \in \mathit{Kind}] \\
\mathit{gmap}\ t & \equiv & \mathit{gmap}'\ \varepsilon\ t
\end{array}
$$

Figure 7.4: Final version of *gmap*

a function from a kind representation to a type. We have not considered type indexed types, but we think that they would fit nicely into this framework.

This encoding falls short of being a semantics for Generic Haskell since it relies on infinite structures not supported by type theory, but even so we think that it gives us a lot of insight into Generic Haskell and can be a useful tool in designing a type system for, or otherwise extending Generic Haskell.

There is still a lot of work that can be done in this area. Obviously the problem of the infinite structures needs to be resolved. Apart from that we want to be able to encode more features of Generic Haskell and formalize the encoding.

Bibliography

[AM02] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In Jeremy Gibbons and Johan Jeuring, editors, *Proceedings of the IFIP WG2.1 Working Conference on Generic Programming 2002*. Kluwer, July 2002. Accepted; to appear. `http://www.dur.ac.uk/c.t.mcbride/generic/`.

[CC99] C. Coquand and T. Coquand. Structured type theory, 1999.

[dW99] Jan de Wit. Implementing polytypic programming for non-regular datatypes, 1999. Avaliable from `http://home.conceptsfa.nl/ jwit/`.

[Hin00] Ralf Hinze. Polytypic values possess polykinded types. In *Mathematics of Program Construction*, volume 1837 of *LNCS*, pages 2–27. Springer-Verlag, 2000.

[HJ] R. Hinze and J. Jeuring. Generic haskell: Practice and theory. To appear in the lecture notes of the Summer School on Generic Programming, LNCS Springer-Verlag, 2002/2003.

[HP01] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science, 2001.

[HR00] Thomas Hallgren and Aarne Ranta. An extensible proof text editor. In *Logic for Programming and Automated Reasoning*, volume 1955 of *LNCS*, pages 70–84. Springer, 2000.

[Jan00] Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden, May 2000.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[PeHeA$^+$99] Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, February 1999.

[PR99] Holger Pfeifer and Harald Rueß. Polytypic proof construction. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proc. 12th Intl. Conf. on Theorem Proving in Higher Order Logics*, number 1690 in LNCS, pages 55–72. Springer-Verlag, 1999.

[Sin01] François-Régis Sinot. Polytypic programming and dependent types, 2001. Project report available from `http://www.ens-lyon.fr/ frsinot/polydep/`.