

A technical overview of Generic HASKELL

Jan de Wit

`jwtit@cs.uu.nl`

Contents

I. Introduction	5
1. Introduction	6
1.1. Abstract	6
1.2. Overview of this thesis	6
1.3. Acknowledgements	6
2. Generic programming	7
2.1. The basic idea of generic programming	7
2.1.1. Defining polytypic values	9
2.2. Types and kinds	11
2.2.1. Kind-indexed types	12
3. Running examples	13
3.1. Datatypes	13
3.2. Type-indexed values	14
3.2.1. Producing and consuming generic values: <i>encode</i> and <i>decode</i> . . .	14
3.2.2. A more complex type: <i>gmap</i>	15
3.2.3. Working with constructors and field labels: <i>show</i>	16
3.3. Type-indexed types	17
3.3.1. Labelling data types	17
3.3.2. Adding labels	18
3.3.3. Extracting labels	19
II. The Generic HASKELL compiler	20
4. Overview of the Generic HASKELL compiler	21
4.1. Overview	21
4.2. The <code>GHPrelude</code>	21
4.2.1. <code>GHPrelude.ghi</code>	21
4.2.2. <code>GHPrelude.hs</code>	22
4.2.3. Extending the <code>GHPrelude</code>	22
4.3. Implementation of the Generic HASKELL compiler	22

5. Preliminaries	23
5.1. Basic definitions	23
5.2. Structure types	24
5.3. Embedding-projection pairs	25
5.4. Bimaps	26
6. Kind-indexed types	28
6.1. Components of a kind-indexed type	28
6.2. The translation of kind-indexed types	29
7. Type-indexed values	31
7.1. Components of a type-indexed value	31
7.2. The translation	32
7.2.1. Translation per definition	32
7.2.2. Translation per data type	32
7.3. An example	34
7.4. Other features	35
7.4.1. Generic applications	35
7.4.2. Generic abstractions	35
8. Type-indexed types	36
8.1. Components of a type-indexed type	36
8.2. The translation	37
8.2.1. Translation per definition	37
8.2.2. Translation per data type	37
8.2.3. Interaction with type-indexed values	38
9. The Generic HVSHELL module system	39
9.1. Goals of the Generic HVSHELL module system	39
9.2. Description of the current implementation	39
III. Using Generic HVSHELL	41
10. The Generic HVSHELL library	42
10.1. Introduction	42
10.2. Overview and naming conventions	42
10.2.1. Generic values and deriving	42
10.3. Module Bounds	43
10.4. Module Collect	43
10.5. Module Compare	44
10.6. Module DeepSeq	44
10.7. Module Eq	45
10.8. Module Labels	45
10.9. Module Map	46

10.10	Module <code>MapM</code>	47
10.11	Module <code>ReadShow</code>	48
10.12	Module <code>Reduce</code>	49
10.13	Module <code>ZipWith</code>	49
11.	Some concrete examples	51
11.1.	A generic structure editor	51
11.1.1.	The basics	51
11.1.2.	The main function	52
11.1.3.	Trying it out	54
11.1.4.	Possible extension	54
11.2.	Other applications of Generic HVSHELL	55
IV.	Concluding words	56
12.	Conclusions and future work	57
12.1.	Current limitations of the Generic HVSHELL compiler	57
12.2.	Experimental extensions	57
12.3.	Future work	58
12.4.	Conclusions	58

Part I.

Introduction

1. Introduction

1.1. Abstract

Generic programming allows programmers to define functions which can operate on a large number of data types, without changing code. **Generic HASKELL** is a superset of Haskell 98 designed specifically for generic programming, adding a number of new features to the Haskell language such as type-indexed values, kind-indexed types and type-indexed types.

This thesis describes these features and the way they currently are translated to Haskell 98 by the **Generic HASKELL** compiler developed at Utrecht University. We also present the **Generic HASKELL** library and a example application of **Generic HASKELL**: a simple generic structure editor, which can be used to interactively construct values of any data type.

1.2. Overview of this thesis

The thesis is divided into four parts: Part I gives a quick overview of generic programming and provides some examples, which will be used later. Part II deals with the internal workings of the **Generic HASKELL** compiler, while Part III gives details about the supplied library and a larger example. Part IV concludes.

1.3. Acknowledgements

First of all, I would like to thank my supervisor Johan Jeuring for giving me the chance to initiate the **Generic HASKELL** project and of course for his helpful comments on this thesis. This thesis also owes much of its existence to Lambert Meertens' wonderful course on generic programming.

I enjoyed working with the other members of the Generic Haskell team: – Dave Clarke, Ralf Hinze and Andres Löh. Thanks to my roommates in room B037, Ade, Arthur, Bastiaan and Karina, and neighbour Jurriaan for on- and off-topic chats and the necessary coffee breaks! Finally, many thanks to Frank Pols and Anet Weterings for their careful reading and commenting of this thesis.

2. Generic programming

This chapter provides a short introduction to Generic Programming as implemented by **Generic HASKELL**. We will give our examples in the function programming language Haskell and we assume that the reader is familiar with Haskell. See the website <http://haskell.org> and [7] for good starting points.

This chapter does not aim to be complete; for more information, including a thorough introduction to generic programming and a solid theoretical background, the reader is referred to [3].

A related approach to generic programming, although with a more limited scope, is described in [2] and [4].

2.1. The basic idea of generic programming

In non-generic programming, a function operating on lists has to be modified or rewritten in order to work on trees. This process of adaptation of code is error-prone and sometimes quite complex. Generic programming is a way of programming that allows the programmer to write functions that can operate on many different data types, without changing code.

As a concrete example, it is a common task to compare two values of the same data type. Since this is a family of functions, each operating on a specific data type, we will denote these functions by eq_T , with the T subscript indicating the type on which the equality function operates. Here is a function that compares two lists containing integers for equality:

$$\begin{aligned} eq_{[Int]} &:: [Int] \rightarrow [Int] \rightarrow Bool \\ eq_{[Int]} [] [] &= True \\ eq_{[Int]} [] (y : ys) &= False \\ eq_{[Int]} (x : xs) [] &= False \\ eq_{[Int]} (x : xs) (y : ys) &= (x == y) \&\& eq_{[Int]} xs ys \end{aligned}$$

Here is one that compares values of type *Maybe Char*.

```

eq_Maybe Char :: Maybe Char → Maybe Char → Bool
eq_Maybe Char Nothing Nothing = True
eq_Maybe Char Nothing (Just d) = False
eq_Maybe Char (Just c) Nothing = False
eq_Maybe Char (Just c) (Just d) = c == d

```

The general pattern should be clear: check whether the values are in the same alternative: if not then they are definitely not equal, otherwise compare the components using the 'appropriate' equality function: the standard `==` function for integers or characters, and – recursively – the `eq[Int]` function for lists of integers.

The concept of using the appropriate equality functions extends naturally to other types. For example, here is the equality function for values of type *Either (Maybe Char) [Int]*:

```

eq_Either (Maybe Char) [Int] :: Either (Maybe Char) [Int] →
                                Either (Maybe Char) [Int] → Bool
eq_Either (Maybe Char) [Int] (Left x) (Left y)
    = eq_Maybe Char x y
eq_Either (Maybe Char) [Int] (Left x) (Right y)
    = False
eq_Either (Maybe Char) [Int] (Right x) (Left y)
    = False
eq_Either (Maybe Char) [Int] (Right x) (Right y)
    = eq[Int] x y

```

If we want to define the equality function on `[Maybe Char]`, `eq[Maybe Char]`, we could reuse the definition of `eq[Int]` by changing `x == y` in the last line to `eq_Maybe Char x y`. This suggests that a general pattern can be obtained by abstracting from the function used to compare `x` and `y`:

```

eq[_] :: (a → a → Bool) → [a] → [a] → Bool
eq[_] eqA [] [] = True
eq[_] eqA [] (y : ys) = False
eq[_] eqA (x : xs) [] = False
eq[_] eqA (x : xs) (y : ys) = (eqA x y) && eq[_] eqA xs ys

```

So the equality function for lists, `eq[_]` turns equality functions for type `a` (functions of type `a → a → Bool`) into equality functions for type `[a]`. The reader can probably guess what `eq_Maybe` looks like. To extend this pattern to *Either*, we need to supply *two*

equality functions:

$$\begin{aligned}
eq_{Either} & :: (a \rightarrow a \rightarrow Bool) \rightarrow (b \rightarrow b \rightarrow Bool) \rightarrow \\
& \quad (Either\ a\ b \rightarrow Either\ a\ b \rightarrow Bool) \\
eq_{Either}\ eqA\ eqB\ (Left\ x)\ (Left\ y) & = eqA\ x\ y \\
eq_{Either}\ eqA\ eqB\ (Left\ x)\ (Right\ y) & = False \\
eq_{Either}\ eqA\ eqB\ (Right\ x)\ (Left\ y) & = False \\
eq_{Either}\ eqA\ eqB\ (Right\ x)\ (Right\ y) & = eqB\ x\ y
\end{aligned}$$

This is a general phenomenon: the type arguments of a data type determine not only the *number* of argument equality functions, but also the *types* of those functions. This is the topic of Section 2.2.

2.1.1. Defining polytypic values

Let us now try to capture the intuition of the preceding section. If we are faced with the task of writing a general equality function for a type T we can proceed as follows:

- Represent the data type as a disjoint sum of a number of types, each corresponding to the product of types inside a constructor.
- In fact, this representation need only use binary disjoint sums and binary products (i.e. 2-tuples).
- At every type, use the 'appropriate' equality function. Note that this can include recursive occurrences as well as some equality functions defined below.

So, first of all, we need to know how to handle disjoint sums of types. We introduce the following notation:

$$\mathbf{data}\ Sum\ a\ b \quad =\ Inl\ a\ |\ Inr\ b$$

We will use $:+:$ as a convenient form for Sum .

If we have two types a and b and equality functions eqA and eqB on these types, we can compute the result of comparing two values in the disjoint sum type $a :+: b$ as follows:

$$\begin{aligned}
eq\ \{\!:\!+\!:\} \quad eqA\ eqB\ x\ y & = \mathbf{case}\ (x, y)\ \mathbf{of} \\
(Inl\ a, Inl\ a') & \rightarrow eqA\ a\ a' \\
(Inl\ a, Inr\ b') & \rightarrow False \\
(Inr\ b, Inl\ a') & \rightarrow False \\
(Inr\ b, Inr\ b') & \rightarrow eqB\ b\ b'
\end{aligned}$$

Binary products of types will be represented by:

```
data Prod a b = a : * : b
```

As with disjoint sums, we will use `:*` instead of `Prod` when writing types. If we are to compare two values in a product type `a : * : b`, given equality functions on `a` and `b`, we get:

```
eq { * : } eqA eqB (a : * : b) (a' : * : b') = eqA a a' && eqB b b'
```

A product of zero types (for constructors without arguments) is analogous to the Haskell unit type `()` and we introduce the following notation:

```
data Unit = Unit
```

For this special case we do not need to supply equality functions or compare values:

```
eq { Unit } x y = True
```

When we are comparing integer (or other built-in types), we can just use the standard equality function:

```
eq { Int } x y = (x == y)
```

If we are comparing two values in the same constructor, we do not need to compare the constructors (they are the same anyway) and just compare the values. However there are a number of functions where more information about the constructor is necessary, for instance when printing or parsing values. We record the fact that a value appears directly inside a constructor as opposed to someplace else by storing the value and a description of the constructor together in a value of this data type:

```
data Con a = Con ConDescr a
```

Here, `ConDescr` is a data type which contains information on a constructor such as its name and its arity. See Section 5.1 for the precise details. The defining case for equality is:

```
eq { Con c } eqA (Con _ a) (Con _ a') = eqA a a'
```

Note that the first occurrence of `Con` is meant to indicate that this definition applies to every value (of type `a`) appearing directly inside a constructor `c`, whereas the second and third occurrences are pattern matches against the `Con` constructor. This is potentially confusing, so the reader is advised to pay attention.

Occasionally, data type declarations can contain field labels, which give a name to a component inside a constructor. Information about these field labels is necessary for printing, but not for equality. The data type used to represent field labels and the code for equality is very similar to the corresponding items for constructors:

```

data Label a = Label LabelDescr a
eqA {Label l} eqA (Label _ a) (Label _ a') = eqA a a'

```

The above definitions are enough to define equality on every data type, since:

- Every disjoint sum of types can be written as a repeated application of the `:+:` operator.
- We can likewise write every tuple using a series of `:*:`'s. The special case of a 0-tuple is covered by the definition for *Unit*.
- If a data type contains the function-space constructor \rightarrow or labelled fields, we can easily add definitions similar to those for `:+:` or *Con* respectively¹.
- Every data type can be represented in this way (see Section 5.2 for more details).

This way of defining equality generically extends naturally to a large number of other functions as well.

2.2. Types and kinds

A concrete data type is a data type that has values in it (possibly only \perp). Examples are *Int*, *String* and so on. The **data** or **type** declarations defining these types have no type arguments.

The type of lists, `[]` is not a data type *per se*, but rather a *type constructor*, which turns a type into another type. It is possible to assign to a type a description of the type arguments it takes, and so we arrive at the concepts of kinds (or 'types of types'). Kinds are written with lowercase Greek letters, starting with κ, λ, \dots

Concrete data types have kind $*$.

A type constructor is a data type which takes one or more type arguments to produce a concrete type. The type of lists turns a concrete type into another concrete type, and we say that it has kind $* \rightarrow *$. *Either* has kind $* \rightarrow * \rightarrow *$, the 3-tuple constructor `(,,)` has kind $* \rightarrow * \rightarrow * \rightarrow *$ and so on.

Type arguments with more complicated kinds are also possible. Take for example the type:

```

data GTree ff a           = GLeaf a
                          | GBranch (ff (GTree ff a))

```

which is the type of trees with labels in the leaves, parameterized over the branching type. If **type** *TwoFork* `a = (a, a)` then *GTree TwoFork* is the binary tree type constructor, but *GTree []* is the type of trees where an internal node has a list of children. *GTree* has kind $(* \rightarrow *) \rightarrow * \rightarrow *$ and we call this a *higher-order* kind.

There are also polymorphic kinds: take for example the data definition:

¹Defining equality for functions is in general not possible.

data $App\ ff\ x = App\ (ff\ x)$

The only restriction on the kinds of ff and x is that ff can take an argument with the kind of x to yield a concrete type, to which the App constructor is applied. So if the kind of x is κ , that of ff is $\kappa \rightarrow *$ and the kind of App itself is $(\kappa \rightarrow *) \rightarrow \kappa \rightarrow *$, for all kinds κ . Haskell does not allow these kind variables: after the kind-inference process any remaining kind variables are replaced by $*$.

We need to introduce two more pieces of terminology: the *order* and the *arity* of kinds.

The order of a kind is defined as follows:

- The order of $*$ is 0.
- The order of $\kappa \rightarrow \lambda$ is the maximum of $order(\kappa) + 1$ and $order(\lambda)$.

So kinds of the form $* \rightarrow \dots \rightarrow *$ have order 1 (or 0 for $*$). The simplest kind with order 2 is $(* \rightarrow *) \rightarrow *$.

The arity of a kind is defined as follows:

- The arity of $*$ is 0.
- The arity of $\kappa \rightarrow \lambda$ is $1 + arity(\lambda)$

In other words, the arity of a kind κ is the number of arguments a type constructor of kind κ needs to have in order to yield a concrete type. The arity of the kind of a data type is directly evident from its definition. The kind of T has arity n if the definition of T looks like:

data $T\ a_1 \dots a_n = \dots$

2.2.1. Kind-indexed types

We can now explain what the exact type of eq is for a type, say T , of kind κ :

- If $\kappa = *$ then

$eq_T :: T \rightarrow T \rightarrow Bool$

We shall refer to this type by the name $Eq\{\{*\}\}\ T$. Note that type arguments of generic functions are surrounded by $\{\}$ and $\}$ whereas kind arguments are surrounded by $\{\{$ and $\}\}$.

- If $\kappa = \lambda \rightarrow \mu$ then eq_T must transform an equality function for an *arbitrary* type u of kind λ into an equality function for type $T\ u$ of kind μ . These considerations lead us to:

$eq_T :: \mathbf{forall}\ u . Eq\{\{\lambda\}\}\ u \rightarrow Eq\{\{\mu\}\}\ (T\ u)$

Put together we have the following:

type $Eq\{\{*\}\}\ t \quad =\ t \rightarrow t \rightarrow Bool$
type $Eq\{\{\kappa \rightarrow \lambda\}\}\ t \quad =\ \mathbf{forall}\ u . Eq\{\{\kappa\}\}\ u \rightarrow Eq\{\{\lambda\}\}\ (t\ u)$

3. Running examples

In this chapter we will present a number of data types, type-indexed values and type-indexed types to be used as running examples.

3.1. Datatypes

We start with the familiar data type of lists. It has kind $* \rightarrow *$.

```
data List a           = Nil
                       | Cons a (List a)
```

LabList is a variant of the *List* data type, with field labels.

```
data LabList a       = LabNil
                       | LabCons{hd :: a, tl :: LabList a }
```

Tree is the type of binary trees with labels in the leaves.

```
data Tree a          = Leaf a
                       | Node (Tree a) (Tree a)
```

Binary is a *non-regular*, or *nested* data type of kind $* \rightarrow *$. It is called non-regular because the type argument of *Binary* inside the constructors is *not* equal to the type argument of the data type. PolyP([4]) is not able to deal with this data type as it is not the fixed point of a functor. This data type can be used to represent lists with efficient item access (see [9])

```
data Binary a        = BNil
                       | BZero (Binary (a, a))
                       | BOne a (Binary (a, a))
```

Rose and *Forest* form a pair of mutually recursive data types.

```
data Rose a          = Rose a (Forest a)
data Forest a        = FNil
                       | FCons (Rose a) (Forest a)
```

GTree, finally, presents an example of a *higher-order data type*. The type argument *ff* is a type-constructor, i.e. it has kind $* \rightarrow *$. The kind of *GTree* is therefore $(* \rightarrow *) \rightarrow * \rightarrow *$.

```

data GTree ff a           = GLeaf a
                          | GBranch (ff (GTree ff a))

```

3.2. Type-indexed values

Below are three examples of type-indexed values:

- The functions *encode* and *decode* which show the basic layout of a generic function definition.
- The function *gmap*. This is a generalization of *map* and *fmap* from the Haskell prelude.
- And finally, *show* is a generic function which makes essential use of constructor names and field labels.

Precisely how the following definitions are translated into normal Haskell code is explained in Part 2.

3.2.1. Producing and consuming generic values: *encode* and *decode*

The function *encode* generically encodes values as a list of bits (which we represent by $[Bool]$). We suppose the existence of a function $primEncodeInt :: Int \rightarrow [Bool]$.

```

type Encode{*} t           = t \rightarrow [Bool]
type Encode{\kappa \rightarrow \lambda} t = forall u. Encode{\kappa} u \rightarrow Encode{\lambda} (t u)
encode\{t :: \kappa\}          :: Encode{\kappa} t

```

In particular, the type of *encode* at kinds $*$ and $* \rightarrow *$ is:

```

encode\{t :: *\}           :: t \rightarrow [Bool]
encode\{t :: * \rightarrow *\} :: (a \rightarrow [Bool]) \rightarrow (t a \rightarrow [Bool])

```

We encode disjoint sums by prefixing with a bit to indicate which compound we are in and binary products are encoded by just concatenating the bits. The *Unit*, *Con* and *Label* cases convey no structural information, and therefore do nothing.

```

encode\{Unit\} Unit       = []
encode\{:+:\} eA eB (Inl a) = False : eA a
encode\{:+:\} eA eB (Inr b) = True  : eB b
encode\{:*:\} eA eB (a:*:b) = eA a ++ eB b
encode\{Con c\} eA (Con _ a) = eA a
encode\{Label l\} eA (Label _ a) = eA a
encode\{Int\} n             = primEncodeInt n

```

Function *decode* takes a list of bits and produces a value and a remaining list of bits (like *ReadS* in the normal Haskell prelude). We wrap the result in the *Maybe* data type to allow for the possibility of failure. Function *decode* is kind of inverse of *encode*. It satisfies the following equalities:

$$\begin{aligned} \text{decode}\{t\} (\text{encode}\{t\} x) &== \text{Just } (x, []) \\ \text{encode}\{t\} (\text{fst } (\text{fromJust } (\text{decode}\{t\} \text{bits}))) &== \text{bits} \end{aligned}$$

provided *bits* is in the range of *encode*, i.e. is a valid encoding of a value of type *t*.

Again, we suppose that *primDecodeInt* :: [Bool] → Maybe (Int, [Bool]) is defined elsewhere. The code for *decode* is similar to that for *encode* apart from wrapping it inside a monadic **do** to deal with failure.

$$\begin{aligned} \text{type Decode}\{*\} t &= [\text{Bool}] \rightarrow \text{Maybe } (t, [\text{Bool}]) \\ \text{type Decode}\{\kappa \rightarrow \lambda\} t &= \text{forall } u. \text{Decode}\{\kappa\} u \rightarrow \text{Decode}\{\lambda\} (t u) \\ \text{decode}\{t :: \kappa\} &:: \text{Decode}\{\kappa\} t \\ \text{decode}\{t :: *\} &:: [\text{Bool}] \rightarrow \text{Maybe } (t, [\text{Bool}]) \\ \text{decode}\{t :: * \rightarrow *\} &:: ([\text{Bool}] \rightarrow \text{Maybe } (a, [\text{Bool}])) \rightarrow [\text{Bool}] \rightarrow \text{Maybe } (t a, [\text{Bool}]) \\ \\ \text{decode}\{\text{Unit}\} \text{bits} &= \text{Just } (\text{Unit}, \text{bits}) \\ \text{decode}\{:+\} \text{decA decB } (\text{False} : \text{bits}) &= \text{do } (a, \text{bits1}) \leftarrow \text{decA } \text{bits} \\ &\quad \text{return } (\text{Inl } a, \text{bits1}) \\ \text{decode}\{:+\} \text{decA decB } (\text{True} : \text{bits}) &= \text{do } (b, \text{bits1}) \leftarrow \text{decB } \text{bits} \\ &\quad \text{return } (\text{Inr } b, \text{bits1}) \\ \text{decode}\{:+\} \text{decA decB } [] &= \text{Nothing} \\ \text{decode}\{:* \} \text{decA decB } \text{bits} &= \text{do } (a, \text{bits1}) \leftarrow \text{decA } \text{bits} \\ &\quad (b, \text{bits2}) \leftarrow \text{decB } \text{bits1} \\ &\quad \text{return } (a:*:b, \text{bits2}) \\ \text{decode}\{\text{Con } c\} \text{decA } \text{bits} &= \text{do } (a, \text{bits1}) \leftarrow \text{decA } \text{bits} \\ &\quad \text{return } (\text{Con } c a, \text{bits1}) \\ \text{decode}\{\text{Label } l\} \text{decA } \text{bits} &= \text{do } (a, \text{bits1}) \leftarrow \text{decA } \text{bits} \\ &\quad \text{return } (\text{Label } l a, \text{bits1}) \\ \text{decode}\{\text{Int}\} \text{bits} &= \text{primDecodeInt } \text{bits} \end{aligned}$$

3.2.2. A more complex type: *gmap*

A common polytypic function is the mapping function, which appears in the Haskell prelude under the names *map* (only for lists) and *fmap*. The generic function, which we shall call *gmap*, has the following type for types *t* of kind ** → **:

$$\text{gmap}\{t :: * \rightarrow *\} :: (a \rightarrow b) \rightarrow t a \rightarrow t b$$

If we try to fit this type into the pattern of kind-indexed types we have seen before, a first try could be:

```

type Map{*} t           = t → t
type Map{κ → λ} t      = forall u. Map{κ} u → Map{λ} (t u)
gmap{t :: κ}            :: Map{κ} t

```

If we check our idea by computing $\text{Map}\{*\ \rightarrow *\} t$, we obtain the type $(a \rightarrow a) \rightarrow t a \rightarrow t a$, which is not general enough.

As explained in more detail in [3][Section 3.3], to get a polytypic mapping function we need a kind-indexed type with *two* type arguments. This allows the argument function to be of type $a \rightarrow b$, while we can ensure that the resulting function is of type $t a \rightarrow t b$ by using t as a parameter to Map twice.

These considerations lead to the following code:

```

type Map{*} t1 t2      = t1 → t2
type Map{κ → λ} t1 t2 = forall u1 u2. Map{κ} u1 u2 → Map{λ} (t1 u1) (t2 u2)
gmap{t :: κ}            :: Map{κ} t t

```

Here is the type of gmap for various kinds:

```

gmap{t :: *}           :: t → t
gmap{t :: * → *}      :: (a → b) → t a → t b
gmap{t :: * → * → *} :: (a → b) → (c → d) → t a c → t b d
gmap{t :: (* → *) → *} :: (forall a b. (a → b) → f a → g b) → t f → t g

```

The definition of gmap is remarkably easy:

```

gmap{Unit} Unit       = Unit
gmap{:+:} gmA gmB x   = case x of
                        Inl a → Inl (gmA a)
                        Inr b → Inr (gmB b)
gmap{:*:} gmA gmB (a:*:b) = (gmA a):*:(gmB b)
gmap{Con c} gmA (Con _ a) = Con c (gmA a)
gmap{Label l} gmA (Label _ a) = Label l (gmA a)
gmap{Int} n             = n

```

3.2.3. Working with constructors and field labels: *show*

The Haskell Report has considerable difficulties in explaining how to derive instances for *Read* and *Show* (see [7][Appendix D.4]). Moreover current Haskell compilers and interpreters usually fail trying to derive instances for these classes when confronted with data types with higher-order kinds. **Generic HASKELL** has no such problems. Presented below is a simple version of *show* in which issues of efficiency and precedence are ignored. Also, data types with labels are not printed correctly. For a more complete implementation of *show* and its inverse *read*, the reader is referred to Section 10.11.

```

type Show{[*]} t           = t → String
type Show{[κ → λ]} t      = forall u . Show{[κ]} u → Show{[λ]} (t u)
show{t :: κ}                :: Show{[κ]} t
show{t :: *}                :: t → String
show{t :: * → *}           :: (a → String) → t a → String

show{Unit} Unit            = ""
show{:+:} shA shB (Inl a)  = shA a
show{:+:} shA shB (Inr b)  = shB b
show{:*:} shA shB (a:*:b)  = shA a ++ "□" ++ shB b
show{Con c} shA (Con _ a)  = showParen True
                             $ showString (conName c) ++ "□" ++ shA a
show{Label l} shA (Label _ a) = showParen True
                             $ showString (labelName l) ++ "=" ++ shA a
show{Int} n                 = show n

```

3.3. Type-indexed types

3.3.1. Labelling data types

It often occurs that we want to decorate a data type with some extra information. For example, if we want to store extra values of type l in the type $Tree$ given above:

```

data Tree a                = Leaf a
                             | Node (Tree a) (Tree a)

```

we obtain the following data type:

```

data TreeL a l             = LeafL l a
                             | NodeL l (TreeL a l) (TreeL a l)

```

We see that every alternative gets an extra argument of type l , and every recursive occurrence of $Tree a$ is replaced by $TreeL$. This operation is easy to perform by hand, and it can also be done by using a *type-indexed type*:

```

type LABELLED{Unit} l     = LUnit Unit
type LABELLED{:+:} lA lB l = LSum (Sum (lA l) (lB l))
type LABELLED{:*:} lA lB l = LProd (Prod (lA l) (lB l))
type LABELLED{Con} lA l   = LCon (l, Con (lA l))
type LABELLED{Label} lA l = LLabel (Label (lA l))
type LABELLED{Int} l      = LInt Int

```

Sums and products are recursively labelled with l 's and only the constructor case (*Con*) gets an extra value of type l^1 .

The next two subsections give two simple type-indexed functions which operate on these labelled data types.

3.3.2. Adding labels

Function *addLabel* adds the same label to every constructor in t .

$$\begin{aligned}
\mathbf{type} \text{ AddLabel}\{*\} t \text{ lab} &= t \rightarrow \text{lab} \rightarrow \text{LABELLED}\{t\} \text{ lab} \\
\mathbf{type} \text{ AddLabel}\{\kappa \rightarrow \lambda\} t \text{ lab} &= \mathbf{forall} \ u. \\
&\quad \text{AddLabel}\{\kappa\} u \text{ lab} \rightarrow \text{AddLabel}\{\lambda\} (t \ u) \text{ lab} \\
\text{addLabel}\{t :: \kappa\} &:: \text{AddLabel}\{\kappa\} t \text{ lab} \\
\text{addLabel}\{t :: *\} &:: t \rightarrow \text{lab} \rightarrow \text{LABELLED}\{t\} \text{ lab} \\
\text{addLabel}\{t :: * \rightarrow *\} &:: (a \rightarrow \text{lab} \rightarrow c \ \text{lab}) \rightarrow \\
&\quad t \ a \rightarrow \text{lab} \rightarrow \text{LABELLED}\{t\} \ c \ \text{lab}
\end{aligned}$$

The c in the type signature for $\text{addLabel}\{t :: * \rightarrow *\}$ appears because the first argument is a label-adding function for an arbitrary type a . This argument would need to have the type $a \rightarrow \text{lab} \rightarrow \text{LABELLED}\{a\} \ \text{lab}$ but since Haskell can not express the relation between a and $\text{LABELLED}\{a\} \ \text{lab}$ we over-generalize the type constructor $\text{LABELLED}\{a\}$ to c . The result type of *addLabel* would be $\text{LABELLED}\{t \ a\} \ \text{lab}$, which gets translated (see Chapter 8) as $\text{LABELLED}\{t\} \ \text{LABELLED}\{a\} \ \text{lab}$.

$$\begin{aligned}
\text{addLabel}\{\text{Unit}\} \ \text{Unit} \ l &= \text{LUnit} \ \text{Unit} \\
\text{addLabel}\{:+\} \ \text{alA} \ \text{alB} \ (\text{Inl} \ a) \ l &= \text{LSum} \ (\text{Inl} \ (\text{alA} \ a \ l)) \\
\text{addLabel}\{:+\} \ \text{alA} \ \text{alB} \ (\text{Inr} \ b) \ l &= \text{LSum} \ (\text{Inr} \ (\text{alB} \ b \ l)) \\
\text{addLabel}\{*\} \ \text{alA} \ \text{alB} \ (a : * : b) \ l &= \text{LProd} \ (\text{alA} \ a \ l : * : \text{alB} \ b \ l) \\
\text{addLabel}\{\text{Con} \ c\} \ \text{alA} \ (\text{Con} \ _ \ a) \ l &= \text{LCon} \ (l, \ \text{Con} \ c \ (\text{alA} \ a \ l)) \\
\text{addLabel}\{\text{Label} \ l'\} \ \text{alA} \ (\text{Label} \ _ \ a) \ l &= \text{LLabel} \ (\text{Label} \ l' \ (\text{alA} \ a \ l)) \\
\text{addLabel}\{\text{Int}\} \ n \ _ &= \text{LInt} \ n
\end{aligned}$$

¹Interestingly, if we replace the 2-tuple type constructor in the *Con* case by *Either*, we get a type-indexed type which is analogous to extending a data type with a variable. The analogy is not complete since *every* alternative gets a variable option, instead of just one.

3.3.3. Extracting labels

Function *splitLabel* retrieves the top-level label from a labelled value, and returns it along with the original value with all labels removed.

```

type SplitLabel{*} t lab      = LABELLED{t} lab → (lab, t)
type SplitLabel{κ → λ} t lab = forall u .
    SplitLabel{κ} u lab → SplitLabel{λ} (t u) lab
splitLabel{t :: κ}           :: SplitLabel{κ} t lab
splitLabel{t :: *}          :: LABELLED{t} lab → (lab, t)
splitLabel{t :: * → *}     :: (c lab → (lab, a)) → LABELLED{t} c lab → (lab, t a)

splitLabel{Unit} (LUnit _) = (noLabel, Unit)
splitLabel{:+:} slA slB (LSum (Inl a)) =
  let (l, a') = slA a
  in (l, Inl a')
splitLabel{:+:} slA slB (LSum (Inr b)) =
  let (l, b') = slB b
  in (l, Inr b')
splitLabel{:*:} slA slB (LProd (a:*:b)) =
  let (_, a') = slA a
    (_, b') = slB b
  in (noLabel, a'*:b')
splitLabel{Con c} slA (LCon (l, (Con _ a))) =
  let (_, a') = slA a
  in (l, Con c a')
splitLabel{Label l} slA (LLabel (Label _ a)) =
  let (_, a') = slA a
  in (noLabel, Label l a')
splitLabel{Int} (LInt (n)) = (noLabel, n)
splitLabel{Char} (LChar (c)) = (noLabel, c)
noLabel = error "no_label_present"

```

Of course the functions *addLabel* and *splitLabel* are not very useful by themselves. For more interesting functions such as *showLabel*, *scanl* and *scanr*, see Chapter 10 on the **Generic HASKELL** Library.

Part II.

The Generic HASKELL compiler

4. Overview of the Generic HVSHELL compiler

4.1. Overview

The **Generic HVSHELL** compiler is a *source-to-source* compiler: it takes a **Generic HVSHELL** source file, processes it and emits a normal Haskell file. The source file can contain:

- Normal Haskell code, possibly containing applications of type-indexed values to types.
- Kind-indexed data types, see Chapter 6.
- Type-indexed values, see Chapter 7.
- Generic applications and abstractions, see Section 7.4.
- Type-indexed types, see Chapter 8.

4.2. The GHPrelude

A compiled **Generic HVSHELL** program needs some basic infrastructure in order to function correctly. The compiler needs to know the kinds of the standard Haskell types and so on, and the compiled program will make use of certain non-standard functions and data types which nevertheless are used in every compiled **Generic HVSHELL** program. The information for the compiler is stored in the `GHPrelude.ghi` file and the non-standard functions and types in the `GHPrelude.hs` module.

4.2.1. GHPrelude.ghi

In order not to have to parse the entire Haskell Prelude each time a **Generic HVSHELL** file is compiled, the following information is stored in the file `GHPrelude.ghi`:

- The kinds of the standard Haskell types, such as `Int`, `[]` and so on.
- The structure type (see Section 5.2) for `Maybe`, `Either`, `[]`, and 2- and 3-tuples.

Furthermore, `GHPrelude.ghi` contains information that will cause bimap (see Section 5.4) to be generated.

4.2.2. `GHPrelude.hs`

Every **Generic HASKELL** source file that is compiled includes the normal Haskell module `GHPrelude.hs`. This file contains a number of things:

- Basic definitions of the types used to represent the structure of user defined data types: *Unit*, *Sum*, *Prod*, *Fun*, *Con* and *Label*. See Section 5.1.
- Types for recording information on constructors and field labels: *ConDescr* and *LabelDescr*. See also Section 5.1.
- Type synonyms for the built-in types (\rightarrow), `[]`, and 2- and 3-tuples : *Fun*, *LIST*, *TUPLE2* and *TUPLE3*, respectively.
- Structure types (see Section 5.2) for the built-in types.
- Basic infrastructure for working with embedding-projection pairs: the data type *EP* and functions *idEP*, *liftEP* and *compE*.
- Embedding-projection pairs (see Section 5.3) for the built-in types, and the *Con* type.
- Bimaps for the built-in types and the structural types (see Section 5.4).

4.2.3. Extending the `GHPrelude`

It is possible to extend the `GHPrelude`, in order to accommodate types located inside libraries. To do this, follow these steps:

- Add the kind of the type to `GHPrelude.ghi`.
- If possible, describe the structure of the type in `GHPrelude.ghi`. Of course, this is not possible for abstract types such as *IO*.
- Manually add a bimap to `GHPrelude.hs`.

4.3. Implementation of the **Generic HASKELL** compiler

The main part of the **Generic HASKELL** compiler is currently implemented as an attribute grammar. The input of this grammar is a parse tree generated by a parser and the output consists of (among others) the generated Haskell file and the Generic Haskell interface file.

5. Preliminaries

5.1. Basic definitions

Every data type in Haskell can be represented as a number of alternatives, each consisting of zero or more components. This is called the *sum-of-products* representation. Each alternative corresponds to a constructor and the components inside of an alternative can be labelled with a field label. To capture this structure faithfully, we need sums, products and information about constructors and field labels. The structural information can be expressed in Haskell as follows:

```
data Sum a b = Inl a | Inr b
data Prod a b = a :* b
data Unit = Unit
```

Note that these types correspond to the pre-defined Haskell type constructors *Either*, *(,)* (2-tuples) and *()* but we define new types to keep them separate.

Information about constructors and field labels is captured in this way:

```
data Con c = Con ConDescr c
data Label f = Label LabelDescr f
```

The values of *ConDescr* or *LabelDescr* describe the constructor or field label that applies, and they are defined like this:

```
data ConDescr = ConDescr
  { conName :: String
  , conType :: String
  , conArity :: Int
  , conLabels :: Bool
  , conFixity :: Fixity
  }
data Fixity
  = Nonfix
  | Infix { prec :: Int }
  | Infixl { prec :: Int }
  | Infixr { prec :: Int }
```



```

type Listo a           = Sum (Con Unit)
                          (Con (Prod a (List a)))
type LabListo a       = Sum (Con Unit)
                          (Con (Prod (Label a)
                                      (Label (LabList a))))

```

As stated above, $List^o$ uses $List$ in its right-hand side.

5.3. Embedding-projection pairs

Since a data type and its structure type are isomorphic, we can generate a pair of functions between them that are inverses of each other:

```

from{T} :: T → To
to{T}   :: To → T

```

These functions satisfy:

```

to . from == id
from . to == id

```

We package these functions in an embedding-projection pair. In this case, the functions contained in this pair are isomorphisms. Embedding-projection pairs have other uses as well; for that reason we choose not to call this data type *Isomorphism*.

```

data EP t to = EP {from :: t → to, to :: to → t}

```

Every type is isomorphic to its structure type; therefore, for any T we can generate:

```

ep{T} :: EP (T a1 ... an) (To a1 ... an)

```

It is fairly straightforward to generate this code for a given T . To go from $T a_1 \dots a_n$ to $T^o a_1 \dots a_n$ we replace each constructor with the correct number of *Inl*'s and *Inr*'s, and for each constructor the type arguments are packed together with a *ConDescr* in the appropriate number of *Prod*'s. They can also be replaced by *Unit* if there are no type arguments. The opposite direction is equally easy. For example:

```

ep{List} :: EP (List a) (Listo a)
ep{List} = EP from to where
  from :: List a → Listo a
  from Nil = Inl (Con conNil Unit)
  from (Cons x xs) = Inr (Con conCons (Prod x xs))
  to :: Listo a → List a
  to (Inl (Con c Unit))
    | c == conNil = Nil
  to (Inr (Con c (Prod x xs)))
    | c == conCons = Cons x xs

```

conNil and *conCons* are the appropriate *ConDescr*'s, see Section 5.1.

The function guards comparing the actual *ConDescr*'s with the descriptions of the constructors from the data type in the *to* function are necessary to ensure that the correct constructors appear at the right places. For example, a user could write a generic function that copies all contents of a value² but changes all constructors into a non-existent constructor. These illegal values are prevented from appearing as the result of this generic function when they are converted back from the structure type. In the current implementation, constructors which do not match or exist can make the generated Haskell program crash without producing an error message.

5.4. Bimaps

When a user-defined data type is used in the type of a type-indexed value it becomes necessary to change values inside the user-defined data type. For example, given the definitions

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
type Poly{[*]} t = t → Tree t
type Poly{[κ → λ]} t = forall u . Poly{[κ]} u → Poly{[λ]} (t u)
poly{t :: κ} :: Poly{[κ]} t
...
```

we can easily generate (by following the process described in Chapter 7) the specializations of *poly* for types T° and S° of kind $*$ and $* \rightarrow *$:

$$poly\{T^\circ\} :: Poly\{[*]\} T^\circ == T^\circ \rightarrow Tree T^\circ$$

and

$$poly\{S^\circ\} :: Poly\{[* \rightarrow *]\} S^\circ == (a \rightarrow Tree a) \rightarrow (S^\circ a \rightarrow Tree (S^\circ a))$$

but we really want to have

$$poly\{T\} :: Poly\{[*]\} T == T \rightarrow Tree T$$

and

$$poly\{S\} :: Poly\{[* \rightarrow *]\} S == (a \rightarrow Tree a) \rightarrow (S a \rightarrow Tree (S a))$$

Converting a T to a T° or vice versa is easy (given $ep\{T\}$) but the more difficult question is how to go from $Tree T^\circ$ to $Tree T$, again given $ep\{T\}$. The problem can be solved if we postulate the existence of a function $bimap\{Tree\}$ with the following type:

²Essentially a variant of *map*, see Section 3.2.2.

$$bimap\{Tree\} :: EP\ a\ b \rightarrow EP\ (Tree\ a)\ (Tree\ b)$$

Given that we also have the following types:

$$\begin{aligned} ep\{T\} &:: EP\ T\ T^\circ \\ poly\{T^\circ\} &:: T^\circ \rightarrow Tree\ T^\circ \end{aligned}$$

We can conclude that:

$$\begin{aligned} bimap\{Tree\}\ ep\{T\} &:: EP\ (Tree\ T)\ (Tree\ T^\circ) \\ to\ (bimap\{Tree\}\ ep\{T\}) &:: Tree\ T^\circ \rightarrow Tree\ T \\ to\ (bimap\{Tree\}\ ep\{T\}) \cdot poly\{T^\circ\} \cdot from\ (ep\{T\}) &:: T \rightarrow Tree\ T \end{aligned}$$

The problem does not occur for *Tree* alone. When a new user-defined data type is introduced, the compiler cannot determine that it will never appear in the base case of a kind-indexed type. This means that functions like $bimap\{Tree\}$ must be generated for *every* data type. This task is fairly easy once we realize that $bimap$ is itself a generic function. It has the following type:

$$\begin{aligned} \mathbf{type}\ Bimap\{*\} t_1 t_2 &= EP\ t_1 t_2 \\ \mathbf{type}\ Bimap\{\kappa \rightarrow \lambda\} t_1 t_2 &= \mathbf{forall}\ u\ v.\ Bimap\{\kappa\} u v \rightarrow Bimap\{\lambda\} (t_1\ u)\ (t_2\ v) \end{aligned}$$

The definition of $bimap$ uses a number of functions defined in `GHPrelude.hs`:

$$\begin{aligned} bimap\{Unit\} &= bimapUnit \\ bimap\{:+:\} &= bimapSum \\ bimap\{:*:\} &= bimapProd \\ bimap\{(\rightarrow)\} &= bimapFun \\ bimap\{Con\ c\} &= bimapCon\ c \\ bimap\{Label\ l\} &= bimapLabel\ l \\ bimap\{Int\} &= bimapInt \end{aligned}$$

Here are two examples, the cases for *Con* and (\rightarrow)

$$\begin{aligned} bimapCon &:: ConDescr \rightarrow EP\ a\ b \rightarrow EP\ (Con\ a)\ (Con\ b) \\ bimapCon\ c\ (EP\ a2b\ b2a) &= EP\{from = (\lambda(Con\ d\ a) \rightarrow Con\ d\ (a2b\ a)) \\ &\quad , to = (\lambda(Con\ d\ b) \rightarrow Con\ d\ (b2a\ b))\} \\ bimapFun &:: EP\ a\ b \rightarrow EP\ c\ d \rightarrow EP\ (a \rightarrow c)\ (b \rightarrow d) \\ bimapFun\ (EP\{from = a2b \\ &\quad , to = b2a\}) \\ &\quad (EP\{from = c2d \\ &\quad , to = d2c\}) \\ &= EP\{from = (\lambda a2c \rightarrow c2d.\ a2c.\ b2a) \\ &\quad , to = (\lambda b2d \rightarrow d2c.\ b2d.\ a2b)\} \end{aligned}$$

So the implementation for $poly\{T\}$ derived above, $to\ (bimap\{Tree\}\ ep\{T\}) \cdot poly\{T^\circ\} \cdot from\ (ep\{T\})$ is in fact equal to $to\ (bimapFun\ ep\{T\})\ (bimap\{Tree\}\ ep\{T\})$.

The $bimap$ function has other uses, as explained in Sections 7.2.2 and 8.2.3.

6. Kind-indexed types

6.1. Components of a kind-indexed type

Consider the type of *gmap*, given in section 3.2.2. The type of $\text{gmap}\{t\}$ varies, depending on t itself and its kind (see section 2.2). A few examples:

$$\begin{array}{ll} \text{gmap}\{Int\} & :: Int \rightarrow Int \\ \text{gmap}\{List\} & :: (a \rightarrow b) \rightarrow (List\ a \rightarrow List\ b) \\ \text{gmap}\{Either\} & :: (a \rightarrow c) \rightarrow (b \rightarrow d) \rightarrow (Either\ a\ b \rightarrow Either\ c\ d) \\ \text{gmap}\{GRose\} & :: (\mathbf{forall}\ x\ y.\ (x \rightarrow y) \rightarrow \mathbf{ff}\ x \rightarrow gg\ y) \rightarrow \\ & (a \rightarrow b) \rightarrow GRose\ \mathbf{ff}\ a \rightarrow GRose\ gg\ b \end{array}$$

The general form of a definition of a kind-indexed type looks like this:

$$\begin{array}{l} \mathbf{type}\ Poly\{\ast\}\ t_1 \dots t_n\ a_1 \dots a_m = T\ t_1 \dots t_n\ a_1 \dots a_m \\ \mathbf{type}\ Poly\{\kappa \rightarrow \lambda\}\ t_1 \dots t_n\ a_1 \dots a_m = \mathbf{forall}\ u_1 \dots u_n.\ \\ Poly\{\kappa\}\ u_1 \dots u_n\ a_1 \dots a_m \rightarrow Poly\{\lambda\}\ (t_1\ u_1) \dots (t_n\ u_n)\ a_1 \dots a_m \end{array}$$

In other words, a kind-indexed type has a number of type parameters varying along with the kind argument and a number of non-varying type parameters.

For a given kind κ , $Poly\{\kappa\}$ is a type constructor. In this example the kind of $Poly\{\kappa\}$ is:

$$\begin{array}{l} Poly\{\kappa\} :: \overbrace{\kappa \rightarrow \dots \rightarrow \kappa}^n \\ \rightarrow \kappa_1 \rightarrow \dots \rightarrow \kappa_m \\ \rightarrow \ast \end{array}$$

In this case the κ_i are the kinds inferred for the a_i from the definition of $Poly\{\ast\}$ for the non-varying type parameters (via the usual kind-inference process, see [7, Section 4.6]).

A few comments can be made here:

- In the current version of the **Generic HASKELL** compiler there is no explicit kind-checking of kind-indexed type definitions.
- The above scheme is not absolutely necessary: the non-varying parameters could change or be permuted, but so far no useful deviations have been found.
- All occurrences of *Poly* must have the same number of arguments. This guarantees that *Poly* is a type constructor with the same number of arguments, regardless of which kind it is specialised to.

6.2. The translation of kind-indexed types

When a kind-indexed type is applied to a kind κ and the correct number of suitably-kinded type arguments, the following happens:

- The definition of the kind-indexed type is expanded.
- The resulting type is converted as much as possible to a normal Haskell type, by removing universal quantifiers. This is done by removing universal quantifiers on the right-hand side of a function arrow (alpha-renaming may be necessary to avoid name-capture) and by removing top-most universal quantifiers. For more detailed information, consult [6].

The rules are:

$$\begin{array}{ll}
 a \rightarrow (\mathbf{forall} \ x . b) & \Rightarrow a \rightarrow b \ (x \notin FV(a)) \\
 \mathbf{forall} \ x . a & \Rightarrow a \ (x \notin FV(a)) \\
 \mathbf{forall} \ x . a & \Rightarrow a \ (\text{only on top level})
 \end{array}$$

For example:

$$\begin{array}{l}
 Poly\{\{*\} \rightarrow * \rightarrow *\} \ t \ a \\
 \equiv \\
 \mathbf{forall} \ u . Poly\{\{*\}\} \ u \ a \rightarrow Poly\{\{*\} \rightarrow *\} \ (t \ u) \ a \\
 \equiv \\
 \mathbf{forall} \ u . T \ u \ a \rightarrow Poly\{\{*\} \rightarrow *\} \ (t \ u) \ a \\
 \equiv \\
 \mathbf{forall} \ u . T \ u \ a \rightarrow (\mathbf{forall} \ v . Poly\{\{*\}\} \ v \ a \rightarrow Poly\{\{*\}\} \ (t \ u \ v) \ a) \\
 \equiv \\
 \mathbf{forall} \ u . T \ u \ a \rightarrow (\mathbf{forall} \ v . T \ v \ a \rightarrow T \ (t \ u \ v)) \\
 \equiv \\
 T \ u \ a \rightarrow (\mathbf{forall} \ v . T \ v \ a \rightarrow T \ (t \ u \ v)) \\
 \equiv \\
 T \ u \ a \rightarrow T \ v \ a \rightarrow T \ (t \ u \ v)
 \end{array}$$

If we instantiate $Poly$ for a higher-order kind such as $(* \rightarrow *) \rightarrow * \rightarrow *$ we obtain:

$$\begin{aligned}
& Poly\{[(*) \rightarrow *) \rightarrow * \rightarrow *]\} t a \\
& \equiv \\
& \mathbf{forall} ff . Poly\{[* \rightarrow *]\} ff a \rightarrow Poly\{[* \rightarrow *]\} (t ff) a \\
& \equiv \\
& \mathbf{forall} ff . (\mathbf{forall} d . Poly\{[*]\} d a \rightarrow Poly\{[*]\} (ff d) a) \rightarrow \\
& \quad (\mathbf{forall} c . Poly\{[*]\} c a \rightarrow Poly\{[*]\} (t ff c) a) \\
& \equiv \\
& \mathbf{forall} ff . (\mathbf{forall} d . T d a \rightarrow T (ff d) a) \rightarrow \\
& \quad (\mathbf{forall} c . T c a \rightarrow T (t ff c) a) \\
& \equiv \\
& (\mathbf{forall} d . T d a \rightarrow T (ff d) a) \rightarrow \\
& \quad (\mathbf{forall} c . T c a \rightarrow T (t ff c) a) \\
& \equiv \\
& (\mathbf{forall} d . T d a \rightarrow T (ff d) a) \rightarrow \\
& \quad T c a \rightarrow T (t ff c) a
\end{aligned}$$

In this case the universal quantifier cannot be eliminated. This is a general pattern: if case for kind $*$ is not constant (i.e. mentions a recursive parameter), a kind-indexed type for a higher-order kind will expand to a type which requires rank-2 polymorphism (or even higher ranks).

7. Type-indexed values

7.1. Components of a type-indexed value

The definition of a type-indexed value follows this pattern:

$$\begin{array}{ll} \mathit{poly}\{t :: \kappa\} & :: \mathit{Poly}\{\{\kappa\}\} t \dots \\ \mathit{poly}\{\mathit{Unit}\} & = f_{\mathit{Unit}} \\ \mathit{poly}\{:+:\} & = f_{\mathit{Sum}} \\ \mathit{poly}\{:*:\} & = f_{\mathit{Prod}} \\ \mathit{poly}\{(\rightarrow)\} & = f_{\mathit{Fun}} \\ \mathit{poly}\{\mathit{Con } c\} & = f_{\mathit{Con}} \\ \mathit{poly}\{\mathit{Label } l\} & = f_{\mathit{Label}} \\ \mathit{poly}\{\mathit{Int}\} & = f_{\mathit{Int}} \\ \dots & \end{array}$$

The lines omitted can also contain *special cases*, overriding the default mechanism of code generation. See the end of section 7.2.1 for more details.

The actual arguments to $\mathit{Poly}\{\{\kappa\}\}$ (in the type of poly) are determined by the number of arguments in the definition of Poly (see Chapter 6).

Furthermore, a case in the definition of a type-indexed value can also have some extra arguments on the left-hand side, as in

$$\mathit{poly}\{T\} a_1 \dots a_n = f_T$$

This is treated in the same way as

$$\mathit{poly}\{T\} = \lambda a_1 \dots a_n \rightarrow f_T$$

except for the fact that when translating this case (see below) the parameters appear on the left-hand side. This is because Hugs currently does not accept rank-2 polymorphic definitions of the form

$$\begin{array}{l} f :: (\mathbf{forall } a \dots) \rightarrow \dots \\ f = \lambda g \rightarrow \dots \end{array}$$

whereas it does accept

$$f g = \dots$$

7.2. The translation

7.2.1. Translation per definition

For each line in the definition of *poly*, a component definition is generated in the Haskell output as follows: the name is changed to the concatenation of *poly* and the typename (e.g. *polyUnit*, *polyInt*). Also, a type signature is added: a case $poly\{\kappa_T\}$ in the definition gets the following type signature (κ_T is the kind of T):

```
polyT :: Poly{\kappa_T} T ...  
polyT = f_T
```

Of course, the type signature is expanded in the generated Haskell file.

There are a number of special cases:

- The components *polyCon* and *polyLabel* take an extra argument of type *ConDescr* and *LabelDescr*. So *polyCon* has type $ConDescr \rightarrow Poly\{*\rightarrow*\} T \dots$
- The components for $:+:$, $:*:$ and (\rightarrow) are named *polySum*, *polyProd* and *polyFun*, respectively.
- If present, the components for $[]$, $(,)$ and $(,)$ are named *polyLIST*, *polyTUPLE2* and *polyTUPLE3*. Normally, function *poly* would be specialised to these types, according to their structure. If these components are present, however, they prevent the generation of code for these types.

The last point is a special case of the principle that user-supplied definitions override code-generation using structure types. So if

```
data Set a = Set [a]  
eq\{Set\} eqA (Set xs) (Set ys) = specialEqCaseForSet xs ys
```

the specialisation

```
eqSet eqA (Set xs) (Set ys) = specialEqCaseForSet xs ys
```

will be generated instead of the standard one, which would strip off the *Set* constructor and compare the lists normally.

7.2.2. Translation per data type

For each user-defined data type

```
data T a1 ... an = C1 t11 ... t1m1  
                | ...  
                | Ck tk1 ... tkmk
```

of kind κ_T two functions are generated:

$polyT :: Poly\{\kappa_T\} T \dots$
 $polyT^\circ :: Poly\{\kappa_T\} T^\circ \dots$

The first specialisation is applicable to the type T and uses the second, which operates on the *structure type* of T , see Chapter 5. The structure type is basically a sum-of-products extended with information about constructors and possibly field labels. The implementation of $polyT^\circ$ follows the structure of T exactly: if

$$\begin{aligned}
T^\circ a_1 \dots a_n &= Sum (Con (Prod t_{11} (Prod \dots t_{1m_1}))) \\
&\quad (Sum (Con (Prod t_{21} (Prod \dots t_{2m_1}))) \\
&\quad \dots \\
&\quad (Con (Prod t_{k1} \dots t_{km_k})))
\end{aligned}$$

then

$$\begin{aligned}
polyT^\circ a_1 \dots a_n &= polySum (polyCon conC_1 (polyProd \overline{t_{11}} \dots \overline{t_{1m_1}})) \\
&\quad \dots \\
&\quad (polyCon conC_k (polyProd \overline{t_{k1}} \dots \overline{t_{km_k}}))
\end{aligned}$$

Here $conC_1, conC_k$ are the *ConDescrs* of C_1 (see Section 5.1). The $\overline{t_{ij}}$ are the translation of the type arguments to the constructors, where type application is translated by application on the value level and type variables are translated to actual variables. We can define the translation scheme from types to values as follows:

- Application: If $T = S U$ then $\overline{T} = \overline{S} \overline{U}$
- Abstraction: If $T = \Lambda a. S$ then $\overline{T} = \lambda \overline{a}. \overline{S}$
- Variable: If $T = a$ then $\overline{T} = gh_a$. (The $gh_$ prefix is basically arbitrary).
- Constant: If $T = C$ where C is a constant type for which $poly$ has a definition, then $\overline{T} = polyC$. (For example, if $T = Int$ then $\overline{T} = polyInt$, if $T = :+:$ then $\overline{T} = polySum$)

For example: the *List* data type has the following structure type (see Section 5.2):

$$\begin{aligned}
\mathbf{type} List^\circ a &= Sum \\
&\quad (Con Unit) \\
&\quad (Con (Prod a (List a)))
\end{aligned}$$

Therefore

$$\begin{aligned}
polyList^\circ gh_a &= polySum \\
&\quad (polyCon conNil polyUnit) \\
&\quad (polyCon conCons (polyProd gh_a (polyList gh_a)))
\end{aligned}$$

Note that $polyList^\circ$ refers to $polyList$. $polyList^\circ$ has type $Poly\{[* \rightarrow *]\} List^\circ$ whereas $polyList$ has type $Poly\{[* \rightarrow *]\} List$. To see how we can obtain the latter type starting from the former we have to take a closer look at the definition of the kind-indexed type $Poly$.

$$\begin{aligned} \mathbf{type} \text{ Poly}\{\{*\}\} t &= \text{Base } t \\ \mathbf{type} \text{ Poly}\{\{\kappa \rightarrow \lambda\}\} t &= \mathbf{forall} \ u . \text{ Poly}\{\{\kappa\}\} \ u \rightarrow \text{ Poly}\{\{\lambda\}\} \ (t \ u) \end{aligned}$$

Here *Base* is a type expression. We will call this the *base case*.

Expanding the definition of *Poly* for $\text{Poly}\{\{*\}\} \text{List}^\circ$ we see that polyList° has type $\text{Base } u \rightarrow \text{Base } (\text{List}^\circ u)$ which we have to transform into a value polyList of type $\text{Base } u \rightarrow \text{Base } (\text{List } u)$. There is no need to change the argument and the result can be obtained by using the function $\text{bimap}\{\text{Base}\} \ \text{ep}\{\text{List}\}$.

In Sections 5.3 and 5.4 the following types are stated:

$$\text{bimap}\{\text{Base}\} :: \text{EP } a \ b \rightarrow \text{EP } (\text{Base } a) \ (\text{Base } b)$$

and

$$\text{ep}\{\text{List}\} :: \text{EP } (\text{List } a) \ (\text{List}^\circ a)$$

so that

$$\begin{aligned} \text{bimap}\{\text{Base}\} \ \text{ep}\{\text{List}\} &:: \text{EP } (\text{Base } (\text{List } a)) \ (\text{Base } (\text{List}^\circ a)) \\ \text{to } (\text{bimap}\{\text{Base}\} \ \text{ep}\{\text{List}\}) &:: \text{Base } (\text{List}^\circ a) \rightarrow \text{Base } (\text{List } a) \end{aligned}$$

Based on this the following can be derived:

$$\text{polyList} = \text{to } (\text{bimap}\{\text{Base}\} \ \text{ep}\{\text{List}\}) \ \text{polyList}^\circ$$

7.3. An example

We obtain the following code if we specialize *gmap* for *List* (see Chapter 3):

$$\begin{aligned} \text{gmapList} &:: (a \rightarrow b) \rightarrow (\text{List } a \rightarrow \text{List } b) \\ \text{gmapList}^\circ &:: (a \rightarrow b) \rightarrow (\text{List}^\circ a \rightarrow \text{List}^\circ b) \\ \text{gmapList} &= \text{to } (\text{bimap}\{\text{Base}\} \ \text{ep}\{\text{List}\} \ \text{ep}\{\text{List}\}) \ \text{gmapList}^\circ \\ \text{gmapList}^\circ \ \text{gh}_a &= \text{gmapSum } (\text{gmapCon } \text{conNil } \text{gmapUnit}) \\ &\quad (\text{gmapCon } \text{conCons } (\text{gmapProd } \text{gh}_a \ (\text{gmapList } \text{gh}_a))) \end{aligned}$$

Here the base case of the kind-indexed type of *gmap* has two parameters so that $\text{bimap}\{\text{Base}\}$ looks like

$$\text{bimap}\{\text{Base}\} \ \text{epA} \ \text{epB} = \text{bimapFun } \text{epA} \ \text{epB}$$

and gets $\text{ep}\{\text{List}\}$ as a parameter twice.

7.4. Other features

7.4.1. Generic applications

Non-polytypic functions in a **Generic HASKELL** program can make use of type-indexed values, by using *generic applications*. These are of the form $poly\{Type\}$, and the usual translation applies. Take, for example, the following **Generic HASKELL** source¹:

```
showInts :: [Int] → String
showInts xs = show\{[Int]\} xs
```

This will be translated to:

```
showInts :: [Int] → String
showInts xs = showLIST showInt xs
```

Generic applications can also occur in class declarations, so that the programmer can write

```
instance Show a ⇒ Show (List a) where
  show xs = show\{List\} show xs
```

7.4.2. Generic abstractions

If a polytypic value has a usage pattern for a specific kind, *generic abstractions* allow the programmer to write this pattern succinctly, without writing a large number of generic applications.

As an example, the following function is true if and only if two data structures have exactly the same shape, and the elements in the first structure are strictly smaller than the elements at corresponding places in the second:

```
f\{t :: * → *\} :: Ord a ⇒ t a → t a → Bool
f\{t\} xs ys = eq\{t\} (<) xs ys
```

This function will be specialised to all data types of kind $* \rightarrow *$ known to the compiler.

¹For the *show* function, see Section 3.2.3

8. Type-indexed types

8.1. Components of a type-indexed type

The definition of a type-indexed type looks like a type-indexed value.

```
type POLY {Unit} t1 ... tn      = TagUnit (...)  
type POLY {:+:} pA pB t1 ... tn  
                                = TagSum (...)  
type POLY {:*} pA pB t1 ... tn  
                                = TagProd (...)  
type POLY {(->)} pA pB t1 ... tn  
                                = TagFun (...)  
type POLY {Con} pA t1 ... tn = TagCon (...)  
type POLY {Label} pA t1 ... tn = TagLabel (...)  
type POLY {Int} t1 ... tn      = TagInt (...)
```

In this definition the *Tag*'s must be valid constructor identifiers which appear nowhere else in the program. Usually they are chosen as the concatenation of *POLY* and *Unit*, *Sum* etc. Between the parentheses on the right-hand side a valid type expression should appear.

POLY does not need to be all uppercase, but we shall use the convention of writing the names of type-indexed types fully capitalized as this helps to distinguish type-indexed types and kind-indexed types.

There are a few differences apart from the keyword **type**:

- The cases for *Con* and *Label* have no argument (like *c* or *l* in the examples of type-indexed value). This is because of the restriction that only types appear on the right-hand side, so there is no way in which the description of the constructor or the label could be used.
- Every case for a type constructor must take a number of arguments which is equal to the arity of the kind of the type constructor (0 for concrete types) and possibly a number of other arguments which should be equal for all cases.
- The types appearing between the parentheses on the right hand side must be kind-correct and fully applied.
- Any uses of *POLY* must have a single argument which is not a type application.

8.2. The translation

8.2.1. Translation per definition

For each line in the definition of the type-indexed type *POLY* a **newtype** declaration appears in the generated Haskell file. These are copied verbatim, except for **newtype** replacing **type** and the removal of the special syntax for *Sum*, *Prod*, *Fun* and possibly [], (,) and (,).

```
newtype POLYUnit  $t_1 \dots t_n$  = TagUnit (...)
newtype POLYSum  $pA\ pB\ t_1 \dots t_n$ 
                                = TagSum (...)
newtype POLYProd  $pA\ pB\ t_1 \dots t_n$ 
                                = TagProd (...)
newtype POLYFun  $pA\ pB\ t_1 \dots t_n$ 
                                = TagFun (...)
newtype POLYCon  $pA\ t_1 \dots t_n$  = TagCon (...)
newtype POLYLabel  $pA\ t_1 \dots t_n$ 
                                = TagLabel (...)
newtype POLYInt  $t_1 \dots t_n$     = TagInt (...)
```

8.2.2. Translation per data type

Suppose we are given a user-defined data type:

```
data  $T\ a_1 \dots a_l$  =  $C_1\ t_{11} \dots t_{1m_1}$ 
                       | ...
                       |  $C_k\ t_{k1} \dots t_{km_k}$ 
```

Then we can compute $POLY\{T\}$ in a manner very similar to the translation process of type-indexed values.

The specialisation $POLY\{T\}^\circ$ for the structure type T° follows the structure exactly:

```
type  $POLYT^\circ\ gha_1 \dots gha_l$  =  $POLYSum\ (POLYCon\ (POLYProd \dots))$ 
                                 $(POLYSum\ (POLYCon\ (POLYProd \dots)))$ 
                                ...
                                 $(POLYCon\ (POLYProd \dots)) \dots$ 
```

The fixed parameters to *POLY* need not appear here, because a type synonym is not required to have a type of kind * on the right-hand side. The translation of $POLY\{T\}$ is just a **newtype** wrapped around $POLYT^\circ$:

```
newtype  $POLYT\ gha_1 \dots gha_l\ t_1 \dots t_n$  =
     $POLYT\ (POLYT^\circ\ gha_1 \dots gha_l\ t_1 \dots t_n)$ 
```

Finally, a trivial embedding-projection pair is generated:

```

epPOLYT :: EP (POLYT gha1 ... ghal t1 ... tn) (POLYTo gha1 ... ghal t1 ... tn)
epPOLYT = let
  from (POLYT x) = x
  to x = POLYT x
  in EP from to

```

8.2.3. Interaction with type-indexed values

Suppose we have a type-indexed value which makes use of the type-indexed type *POLY*.

```

type ToPoly{*} t           = t → POLY {t}
type ToPoly{κ → λ} t      = forall u. ToPoly{κ} u → ToPoly{λ} (t u)

toPoly {t :: κ}           :: ToPoly{κ} t
toPoly {Unit} = ...
...

```

We want to specialise this value to *List*. The type of *toPoly* is like the function in Section 5.4 where the normal type *Tree* has been replaced with the type-indexed type *POLY*. We can use the code generation process outlined in Subsection 7.2.2. For *toPolyList^o* this results in:

```

toPolyListo :: (u → uPOLY) → (Listo u → POLYListo uPOLY)
toPolyListo gh_a = toPolySum (toPolyCon conNil toPolyUnit)
  (toPolyCon conCons (toPolyProd gh_a) (toPolyList gh_a))

toPolyList :: (u → uPOLY) → (List u → POLYList uPOLY)
toPolyList gh_a = to (bimapFun epList epPOLYList) (toPolyListo gh_a)

```

9. The Generic HVSHELL module system

9.1. Goals of the Generic HVSHELL module system

As has been demonstrated in more detail in previous chapters, when specializing generic definitions, the **Generic HVSHELL** compiler has to generate code for a number of different purposes.

For instance, when specializing a type-indexed value *poly* to a specific data type T , we need to have the following pieces of code:

- The structure type T° of T .
- The embedding-projection pair establishing the isomorphism between T and T° .
- The bimap for T and T° .
- The code for $polyT$ and $polyT^\circ$.

When we are writing library code, we usually give only the polytypic value definition and not the types at which it is used. We also want to be able to override the polytypic definition (e.g. when defining equality for sets, see Section 7.2.1).

We can not anticipate when this is necessary.

The goals of the **Generic HVSHELL** module system are as follows:

- Allow separation of polytypic definitions and the data types at which they are to be used.
- Allow extension or overriding of polytypic definitions.
- Generation of as little code as possible.

9.2. Description of the current implementation

The module system of **Generic HVSHELL** has not yet been the subject of much research so we currently take the following *ad hoc* approach. This is certain to change in the future.

In order to generate code at most once, we generate the code specific to the data type and the polytypic definition only for the **Generic HVSHELL** files where they are defined. We deal with extensions to polytypic definitions in separate files by generating the requested line in the compiled file. The question of when to generate code for a specialization of a definition to a data type is more difficult. Currently we require that

either the data type or the polytypic definition be local (i.e. defined in the module being compiled) in order to generate code.

Example: suppose `A.ghs` contains the definition of the polytypic value f , `B.ghs` contains the data type T and `C.ghs` contains both a polytypic value g and a data type S and imports `A.ghs` and `B.ghs`. Then when we compile `C.ghs`, the generated file `C.hs` will contain:

- The g - and S - specific code (structure types etc.)
- The specialization of g to S
- The specialization of g to T
- The specialization of f to S
- But *not* the specialization of f to T

(`A.hs` will only contain the code specific to f and `B.hs` will contain the type T and code related to it)

There are several drawbacks to this approach:

- Code is specialized whether or not it is needed.
- There is no way to get f specialized to T in the example above.

Part III.

Using Generic HASKELL

10. The Generic HASKELL library

10.1. Introduction

The **Generic HASKELL** system comes with a library of useful generic functions. These are summarised in the following sections; for the implementation, consult the library itself. We give the kind-indexed types of the generic functions, show what these types are when specialised to kind `*` and `* → *`, and a short description. In a number of cases we also give a more formal account of the properties of the function, such as laws that hold for the functions at kind `*` or `* → *`. We do not do this in full generality, as this would need the notion of *logical relations*, see [3][Section 4.3]. The properties stated below can be overruled by overriding the standard generic definition. For example

$$eq\{t\} x x == True$$

does not necessarily hold for all x of type t , in the presence of a definition

$$eq\{T\} _ _ = False$$

We shall omit explicitly mentioning these exceptional cases.

10.2. Overview and naming conventions

When generic functions defined in the **Generic HASKELL** library have an equivalent in the Haskell Prelude or libraries, the name of the generic function is prefixed with a ‘g’.

10.2.1. Generic values and deriving

The following standard Haskell typeclasses are derivable via the **deriving** construct: *Eq*, *Ord*, *Enum*, *Bounded*, *Show*, *Read* and the Haskell Library class *Ix*. See [7][Appendix D] and [8][Section 5]. The **Generic HASKELL** Library currently provides equivalent type-indexed values for all of these type classes, except for *Enum* and *Ix*. In addition, a generalization of the *fmap* function in the *Functor* class is provided.

10.3. Module Bounds

```

type Bounds{*} t                = t
type Bounds{κ → λ} t           = forall u . Bounds{κ} u → Bounds{λ} (t u)
gminBound, gmaxBound{t :: κ}    :: Bounds{κ} t
gminBound, gmaxBound{t :: *}    :: t
gminBound, gmaxBound{t :: * → *} :: a → t a

```

Functions *gminBound* and *gmaxBound* are generalizations of the *minBound* and *maxBound* members of the *Bounded* class. These functions are also defined for types for which *Bounded* is not derivable; i.e. types which are not enumerations or simple product types (see [7, Appendix D]).

These functions have the property that for all types *t* of kind ***:

```

∀ a :: t : gcompare{t} (gminBound{t}) a ∈ [LT, EQ]
∀ a :: t : gcompare{t} a (gmaxBound{t}) ∈ [LT, EQ]

```

In words: *gminBound{t}* is the smallest value of type *t* while *gmaxBound{t}* is the largest, where the order is the 'standard' generic one. As usual, this fact only holds if *gcompare* and *gminBound* or *gmaxBound* do not have overriding cases.

10.4. Module Collect

The functions in this module collect information about types and values of these types. Their types follow some general patterns which are defined in this file as well.

```

type Collect0{*} a              = a
type Collect0{κ → λ} a          = Collect0{κ} a → Collect0{λ} a
type Collect{*} t a             = t → a
type Collect{κ → λ} t a        = forall u . Collect{κ} u a → Collect{λ} (t u) a

```

Function *constructorOf* returns a description of the topmost constructor of a value.

```

constructorOf{t :: κ}           :: Collect{κ} t ConDescr
constructorOf{t :: *}          :: t → ConDescr
constructorOf{t :: * → *}      :: (a → ConDescr) → t a → ConDescr

```

Function *constructors* returns a list of descriptions of all topmost constructors used in a data type.

```

constructors{t :: κ}           :: Collect0{κ} [ConDescr]
constructors{t :: *}          :: [ConDescr]
constructors{t :: * → *}      :: [ConDescr] → [ConDescr]

```

Function *labelsOf* returns a list of descriptions of labels in a value or the empty list when the current type constructor has no labels.

```

labelsOf {t :: κ} :: Collect {κ} t [LabelDescr]
labelsOf {t :: *} :: t → [LabelDescr]
labelsOf {t :: * → *} :: (a → LabelDescr) → t a → LabelDescr

```

The function *constructorsAndLabels* combines the above information: it returns a list of all constructors paired with the labels present in the given type constructor.

```

constructorsAndLabels {t :: κ} :: Collect0 {κ} [(ConDescr, [LabelDescr])]
constructorsAndLabels {t :: *}      :: [(ConDescr, [LabelDescr])]
constructorsAndLabels {t :: * → *}  :: [(ConDescr, [LabelDescr])]
                                       → [(ConDescr, [LabelDescr])]

```

Consult `GHPrelude.hs` or section 5.1 for details of *ConDescr* and *LabelDescr*.

10.5. Module Compare

```

type Compare {[*]} t1 t2      = t1 → t2 → Ordering
type Compare {κ → λ} t1 t2   = forall u1 u2 .
                               Compare {κ} u1 u2 → Compare {λ} (t1 u1) (t2 u2)

```

Function *gcompare* is the generic version of *compare* in the *Ord* class.

```

gcompare {t :: κ}              :: Compare {κ} t t
gcompare {t :: *}              :: t → t → Ordering
gcompare {t :: * → *}          :: (a → b → Ordering) → t a → t b → Ordering

```

It satisfies the following laws:

```

gcompare {t} x x ≡ EQ
gcompare {t} x y ≡ EQ ⇒ gcompare {t} y x ≡ EQ
gcompare {t} x y ≡ LT ⇒ gcompare {t} y x ≡ GT
gcompare {t} x y ≡ LT && gcompare {t} y z ≡ LT ⇒ gcompare {t} x z ≡ LT
gcompare {t} x y ≡ EQ ⇔ eq {t} x y ≡ True

```

10.6. Module DeepSeq

```

type DSeq {[*]} t           = t → b → b
type DSeq {κ → λ} t        = forall u . DSeq {κ} u → DSeq {λ} (t u)

```

dSeq is a generalization of the function *seq* (with type $a \rightarrow b \rightarrow b$) from the Haskell Prelude.

```

dSeq{ $t :: \kappa$ }           :: DSeq{ $\kappa$ } t
dSeq{ $t :: *$ }           ::  $t \rightarrow b \rightarrow b$ 
dSeq{ $t :: * \rightarrow *$ } ::  $(a \rightarrow b \rightarrow b) \rightarrow (t a \rightarrow b \rightarrow b)$ 

```

Function *seq* forces evaluation of its first argument, but only far enough to see whether it is \perp or not; if it is \perp then *seq* returns \perp as well, otherwise its second argument.

dSeq evaluates its argument fully. Example: if *xs* is a *List* of *Ints*, *seq xs* evaluates it until the top level constructor is known (*Nil*, *Cons* or \perp), *dSeqList dSeqInt xs* or *dSeqList seq xs* fully evaluates the list and *dSeqList (const id) xs* evaluates only the structure of the list.

10.7. Module Eq

```

type Eq{ $*$ } t1 t2           =  $t_1 \rightarrow t_2 \rightarrow Bool$ 
type Eq{ $\kappa \rightarrow \lambda$ } t1 t2 = forall u1 u2 .
                                     Eq{ $\kappa$ } u1 u2  $\rightarrow$  Eq{ $\lambda$ } (t1 u1) (t2 u2)

```

Function *eq* is the generic version of (*==*) in the *Eq* class.

```

eq{ $t :: \kappa$ }           :: Eq{ $\kappa$ } t t
eq{ $t :: *$ }           ::  $t \rightarrow t \rightarrow Bool$ 
eq{ $t :: * \rightarrow *$ } ::  $(a \rightarrow b \rightarrow Bool) \rightarrow t a \rightarrow t b \rightarrow Bool$ 

```

It satisfies the following laws:

```

eq{ $t$ } x x  $\equiv$  True
eq{ $t$ } x y  $\equiv$  eq{ $t$ } y x
eq{ $t$ } x y && eq{ $t$ } y z  $\Rightarrow$  eq{ $t$ } x z

```

10.8. Module Labels

```

type LABELLED{Unit} l           = LUnit Unit
type LABELLED{ $:+:$ } lA lB l       = LSum (Sum (lA l) (lB l))
type LABELLED{ $:*$ } lA lB l       = LProd (Prod (lA l) (lB l))
type LABELLED{ $(\rightarrow)$ } lA lB l   = LFun (lA l  $\rightarrow$  lB l)
type LABELLED{Con} lA l        = LCon (l, Con (lA l))
type LABELLED{Label} lA l      = LLabel (Label (lA l))
type LABELLED{Int} l          = LInt Int
type LABELLED{Char} l         = LChar Char

```

The type-indexed type $LABELLED\{t\}$ l is the same type as t , with every constructor decorated with a label of type l . $LABELLED$ lifts the kind of its type argument, in the sense that in the kind of t every $*$ is replaced by $(* \rightarrow *)$. For example, $LABELLED\{List\}$ has kind $(* \rightarrow *) \rightarrow * \rightarrow *$; the first type argument indicates how the original values in $List$ are to be labelled, and the second type argument is the type of the label per constructor.

The following functions operate of values of this type:

```

type AddLabel $\{*\}$   $t$   $lab = t \rightarrow lab \rightarrow LABELLED\{t\}$   $lab$ 
type AddLabel $\{k \rightarrow l\}$   $t$   $lab = \mathbf{forall}$   $u$  .
    AddLabel $\{k\}$   $u$   $lab \rightarrow AddLabel\{l\}$   $(t\ u)$   $lab$ 
addLabel $\{t :: k\} :: AddLabel$        $\{k\}$   $t$   $lab$ 
addLabel $\{t :: *\}$   $:: t \rightarrow lab \rightarrow LABELLED(t)$   $lab$ 
addLabel $\{t :: * \rightarrow *\}$   $:: (a \rightarrow lab \rightarrow c\ lab) \rightarrow t\ a \rightarrow lab \rightarrow LABELLED(t)$   $c\ lab$ 

```

$addLabel\{t\}$ $x\ l$ adds a fixed label l to $x :: t$. When using the function for kinds other than $*$, there are several possibilities for the first argument. For example $(,)$ to combine the original value with the label or $const$. $Left$ to keep only the first original value. $const$, of type $a \rightarrow b \rightarrow a$ cannot be used because the Haskell type-checker is unable to determine that c could be $\Lambda\ l . a$.

```

type SplitLabel $\{*\}$   $t$   $lab = LABELLED\{t\}$   $lab \rightarrow (lab, t)$ 
type SplitLabel $\{k \rightarrow l\}$   $t$   $lab = \mathbf{forall}$   $u$  .
    SplitLabel $\{k\}$   $u$   $lab \rightarrow SplitLabel\{l\}$   $(t\ u)$   $lab$ 
splitLabel $\{t :: k\} :: SplitLabel\{k\}$   $t$   $lab$ 

```

Function $splitLabel$ separates a labelled value into the top-level label and the unlabelled value. The following law holds:

$$splitLabel\{t\} (addLabel\{t\} x\ l) \equiv (l, x)$$

10.9. Module Map

```

type Map $\{*\}$   $t_1$   $t_2 = t_1 \rightarrow t_2$ 
type Map $\{\kappa \rightarrow \lambda\}$   $t_1$   $t_2 = \mathbf{forall}$   $u_1\ u_2$  .
    Map $\{\kappa\}$   $u_1\ u_2 \rightarrow Map\{\lambda\}$   $(t_1\ u_1)$   $(t_2\ u_2)$ 

```

Function $gmap$ is the generic version of map and its generalization $fmap$ in the *Functor* class.

```

gmap $\{t :: \kappa\}$        $:: Map\{\kappa\}$   $t\ t$ 
gmap $\{t :: *\}$        $:: t \rightarrow t$ 
gmap $\{t :: * \rightarrow *\}$   $:: (a \rightarrow b) \rightarrow t\ a \rightarrow t\ b$ 

```

The following law applies when t has kind $*$:

$$gmap\{t\} \equiv id$$

The following laws apply when t has kind $* \rightarrow *$:

$$\begin{aligned} gmap\{t\} id &\equiv id \\ gmap\{t\} f . gmap\{t\} g &\equiv gmap\{t\} (f . g) \end{aligned}$$

The reader may recognise these as the the functor laws.

10.10. Module MapM

$$\begin{aligned} \text{type } MapM\{\ast\} t_1 t_2 m &= t_1 \rightarrow m t_2 \\ \text{type } MapM\{\kappa \rightarrow \lambda\} t_1 t_2 m &= \text{forall } u_1 u_2 . \\ &MapM\{\lambda\} u_1 u_2 m \rightarrow MapM\{\kappa\} (t_1 u_1) (t_2 u_2) m \end{aligned}$$

Functions $mapMl$ and $mapMr$ are the generic versions of the monadic map $mapM$ in the Prelude.

$$\begin{aligned} mapMl, mapMr\{t :: \kappa\} &:: (Functor m, Monad m) \Rightarrow MapM\{\kappa\} t t m \\ mapMl, mapMr\{t :: *\} &:: (Functor m, Monad m) \Rightarrow t \rightarrow m t \\ mapMl, mapMr\{t :: * \rightarrow *\} &:: (Functor m, Monad m) \Rightarrow (a \rightarrow m b) \rightarrow t a \rightarrow m (t b) \end{aligned}$$

$mapMl$ traverses a data structure from left to right (just like $mapM$) while $mapMr$ traverses from right to left¹.

The following law applies when t has kind $*$:

$$mapMl\{t\} \equiv mapMr\{t\} == return$$

The following laws apply when t has kind $* \rightarrow *$:

$$\begin{aligned} mapMl\{t\} return &\equiv mapMr\{t\} return \equiv id \\ (mapMl\{t\} f xs) \gg= mapMl\{t\} g &\equiv mapMl\{t\} (\lambda x \rightarrow f x \gg= g) xs \end{aligned}$$

The Haskell Prelude function $mapM$ is a special case of $mapMl$; the following laws hold:

$$\begin{aligned} mapMl\{[]\} &= mapM \\ mapMr\{[]\} &= mapM . reverse \end{aligned}$$

¹The reason for mentioning *Functor* in the context is that in Haskell'98, *Functor* is not a superclass of *Monad*.

10.12. Module Reduce

```

type RReduce{[*]} t b           = t → b → b
type RReduce{[κ → λ]} t b     = forall u .
                                RReduce{[κ]} u b → RReduce{[λ]} (t u) b

rreduce{t :: κ}                 :: RReduce{[κ]} t b
rreduce{t :: *}                 :: t → b → b
rreduce{t :: * → *}            :: (a → b → b) → t a → b → b

type LReduce{[*]} t b           = b → t → b
type LReduce{[κ → λ]} t b     = forall u .
                                LReduce{[κ]} u b → LReduce{[λ]} (t u) b

lreduce{t :: κ}                 :: LReduce{[κ]} t b
lreduce{t :: *}                 :: b → t → b
lreduce{t :: * → *}            :: (b → a → b) → b → t a → b

```

rreduce is a generic version of *foldr* (note the reversed order of the last two arguments!), while *lreduce* is a generic *foldl*. See [3, section 5.4] and [1].

```
crush{t :: * → *} :: (a → a → a) → a → t a → a
```

crush is an instance of *lreduce*.

The following functions are all defined in terms of the above functions, and most have counterparts in the Haskell Prelude:

```

gsum, gproduct{t :: * → *}      :: Num a ⇒ t a → a
gand, gor{t :: * → *}           :: t Bool → a
flatten{t :: * → *}             :: t a → [a]
count{t :: * → *}               :: t a → Int
comp{t :: * → *}                :: t (a → a) → (a → a)
gconcat{t :: * → *}             :: t [a] → [a]
gall, gany{t :: * → *}          :: (a → Bool) → t a → Bool
gelem{t :: * → *}               :: Eq a ⇒ a → t a → Bool

```

flatten collects all values of type *a* in a list, and *comp* composes all functions contained in a data type.

10.13. Module ZipWith

```

type ZipWith{[*]} t1 t2 t3 = (t1, t2) → Maybe t3
type ZipWith{[κ → λ]} t1 t2 t3 = forall u1 u2 u3 .
                                ZipWith{[κ]} u1 u2 u3 → ZipWith{[λ]} (t1 u1) (t2 u2) (t3 u3)

gzipWith{t :: κ}                :: ZipWith{[κ]} t t t
gzipWith{t :: *}                :: (t, t) → Maybe t
gzipWith{t :: * → *}            :: ((a, b) → Maybe c) → (t a, t b) → Maybe (t c)

```

Function *gzipWith* is a generic version of *zipWith*. It differs from *zipWith* in that it returns *Nothing* when the two values of the data types do not have the same shape, whereas *zipWith* returns [] in that case.

```

type UnzipWith{[*]} t1 t2 t3    = t1 → (t2, t3)
type UnzipWith{[κ → λ]} t1 t2 t3 = forall u1 u2 u3 .
                                         UnzipWith{[κ]} u1 u2 u3 → UnzipWith{[λ]} (t1 u1) (t2 u2) (t3 u3)

gzipWith{t :: κ}           :: UnzipWith{[κ]} t t t
gzipWith{t :: *}          :: t → (t, t)
gzipWith{t :: * → *}     :: (a → (b, c)) → t a → (t b, t c)

```

Function *gunzipWith* is a generic version of *unzip*.

This law holds when $t :: *$:

$$\text{gunzipWith}\{t\} x \equiv (x, x)$$

The following law holds when $t :: * \rightarrow *$:

$$\text{gunzipWith}\{t\} \text{id } as \equiv (bs, cs) \Rightarrow \text{gzipWith } \text{Just } (bs, cs) \equiv \text{Just } as$$

The following functions are more or less direct generalizations of *zip* and *unzip* respectively and are defined as instances of *gzipWith* and *gunzipWith*.

```

gzip{t :: * → *} :: t a → t b → Maybe (t (a, b))
gunzip{t :: * → *} :: t (a, b) → (t a, t b)

```

11. Some concrete examples

11.1. A generic structure editor

As an example of a slightly more complicated **Generic HASKELL** program, we present a generic structure editor. This editor lets the user interactively construct values of any type. This example shows a way to deal with information flows that are more complicated than the library functions.

11.1.1. The basics

We use the following data as input to perform the editing action:

- The constructors of the type, for showing to the user at the `:+:` alternative.
- The context of the value being edited, i.e. the result of showing the value constructed so far.

```
data EditIn t = EditIn
  { constrsIn :: [ConDescr]
  , contextIn :: String    -- EditContext
  } deriving Show
```

When editing a value we collect the following information:

- The constructors of the data type of which we are constructing a value.
- The resulting value and context. Since these are dependent on user input, they are wrapped-up inside the *IO* monad.

```
data EditOut t = EditOut
  { action :: IO (EditAction t)
  , constrsOut :: [ConDescr]
  } deriving Show

data EditAction t = EditAction
  { valueOut :: t
  , contextOut :: String    -- EditContext
  } deriving Show
```

The type of the main function is given by the following kind-indexed type:

```

type Edit{*} t = EditIn t → EditOut t
type Edit{κ → λ} t = forall u . Edit{κ} u → Edit{λ} (t u)
edit{t :: κ} :: Edit{κ} t

```

11.1.2. The main function

Editing the unit type requires no input from the user:

```

edit{Unit} ei = EditOut (return (EditAction Unit (contextIn ei))) []

```

Editing a sum means that the user has to decide which constructor is to be used. We print all constructors and ask whether the current left alternative is to be used. The result of the editing action is the value and context of the corresponding alternative.

```

edit{| :+ : |} editA editB ei = let
  eiA = EditIn (constrsIn ei) (contextIn ei)
  eiB = EditIn (constrsIn ei) (contextIn ei)
  eoA = editA eiA
  eoB = editB eiB
  constrsOut' = constrsOut eoA ++ constrsOut eoB
  action' = do
    putStr "Context:␣"
    putStrLn $ contextIn ei
    putStr "All␣constructors:␣"
    print (constrsIn ei)
    putStr "Do␣you␣want␣the␣"
    putStr (show (head (constrsOut eoA)))
    putStr "␣constructor␣(y/n,␣default␣=␣n)?␣"
    s ← getLine
    if head (s ++ "n") == 'y'
    then do
      actionA ← action eoA
      return (EditAction (Inl (valueOut actionA)) (contextOut actionA))
    else do
      actionB ← action eoB
      return (EditAction (Inr (valueOut actionB)) (contextOut actionB))
in EditOut action' constrsOut'

```

Editing a product simply performs the two actions after each other; the incoming context of the second editing action is the outgoing context of the first.

```

edit{| :* : |} editA editB ei = let
  eiA0 = EditIn [] (contextIn ei)
  eoA0 = editA eiA0
  eiB0 = EditIn [] (contextIn ei)

```

```

eoB0 = editB eiB0
eiA = EditIn (constrsOut eoA0) (contextIn ei)
eoA = editA eiA
action' = do
  actionA ← action eoA
  let ciB = contextOut actionA ++ "␣"
  let eiB = EditIn (constrsOut eoB0) ciB
  let eoB = editB eiB
  actionB ← action eoB
  let voA = valueOut actionA
  let voB = valueOut actionB
  return (EditAction (voA:*:voB) (contextOut actionB))
in EditOut action' []

```

Editing a constructor involves adding the constructor to the context.

```

edit{ | Con c | } editA ei = let
  ciA = contextIn ei ++ "(" ++ show c ++ "␣"
  eiA = EditIn [] ciA
  eoA = editA eiA
  action' = do
    putStr "Context:␣"
    putStrLn $ contextIn ei
    putStr "\nCurrent␣alternative:"
    print c
    actionA ← action eoA
    let vo = (Con c (valueOut actionA))
    putStrLn $ "done␣constructing␣a␣" ++ (show c) ++ "␣of␣type␣" ++ (conType c)
    return (EditAction vo (contextOut actionA ++ "))")
in EditOut action' [c]

```

Editing values of built-in type just asks for a value of that type until a valid one is entered.

```

edit{ | Int | } = editStandard "Int"
edit{ | Char | } = editStandard "Char"
edit{ | Bool | } = editStandard "Bool"
edit{ | String | } = editStandard "String"
editStandard :: (Read a, Show a) ⇒ String → EditIn a → EditOut a
editStandard s ei = let
  constrsOut = []
  action = do
    putStr "Context:␣"

```

```

    putStrLn $ contextIn ei
    putStr $ "Enter new value of type " ++ s ++ ":"
    s ← getLine
    let res = reads s
    if (null res)
      then putStrLn "Parse error!" >> action
      else return $ EditAction (fst (head res)) (contextIn ei ++ s)
  in EditOut action constraOut

```

11.1.3. Trying it out

The editing function which is called first needs to have the information about the constructors. The *test* function first runs the editing action without this information and then extracts only the constructors which are used in the second run. Laziness guarantees that the first action is not performed so that no input is requested from the user.

```

test :: (EditIn a → EditOut a) → IO a
test edit = do
  let ei = EditIn [] (editContext "")
      eo = edit ei
      ei2 = EditIn (constraOut eo) (editContext "")
      eo2 = edit ei2
      a ← action eo2
      let v = valueOut a
  return v

```

The *main* function could look like this:

```

main :: IO ()
main = do
  t ← test (edit{| Tree Int |})
  l ← test (edit{| List Char |})
  print t
  print l

```

11.1.4. Possible extension

Possible extensions to this program include:

- A friendlier user interface, where for example the user can type the name of the required constructor instead of answering ‘no’ to the question ‘Do you want this constructor?’ a number of times. Even a graphical user interface is possible.
- Allowing the user to edit values as well as construct them.
- The possibility of backtracking in the editing process.

11.2. Other applications of Generic HASKELL

Since the **Generic HASKELL** compiler has not been around for very long, there are not many applications yet. Worth mentioning however, is the XML-compressor developed by Jeuring, and described in [5]. Initial tests indicate that this XML-compressor performs considerably better (25% to 40%) than other XML-compressors and has a much smaller code size.

XML seems to be a fertile field for applications of **Generic HASKELL**, since an XML Data Type Definition essentially corresponds to a Haskell data type of kind `*`.

Part IV.

Concluding words

12. Conclusions and future work

12.1. Current limitations of the Generic HASKELL compiler

The first experiences with the **Generic HASKELL** compiler at Utrecht University show that kind-indexed types and type-indexed values work as expected.

It is not possible to write polytypic functions which use each other¹ in an easy way. For example, the *edit* function defined in Chapter 11 also performs the task of the *constructors* function defined in Section 10.4. For writing polytypic functions which depend on each other the so-called ‘MPC’ style (see [3][Chapter 3]) of generic definitions may be better suited.

The way the module system currently works is a serious impediment to the development of large-scale programs.

12.2. Experimental extensions

Already, a number of experimental extensions have been added to the **Generic HASKELL** compiler:

- It is possible to override type-indexed values for specific constructors. For example, one could write:

```
gmap{f} case Cons{f} (Cons x xs) = Cons (f x) (Cons (f x) (gmap{List} f xs))
```

This would have the effect that when *gmap* is used on values of type *List*, all elements would be duplicated. The built-in type of list and other list-like types would continue to behave normally.

The ability to do this would be useful in a variety of situations, such as modifying *show* to print certain expressions differently.

- A simple way of altering a type-indexed value is provided by the *copy line* mechanism. As an example take the following modification of the *gmap* function:

```
addMap{t ::  $\kappa$ } :: Map{ $\kappa$ } t t
```

```
addMap{Int} n = n + 1
```

```
addMap{c} = gmap{c}
```

The last line causes the compiler to copy all the definitions from the *gmap* function, except for the one overridden above. Therefore the function *addMap* works just

¹The functions do not even have to be mutually recursive, as the example of *edit* and *constructors* shows.

like *gmap* except that it adds 1 to all integers in a data structure.

12.3. Future work

From a usability perspective, a lot of work remains to be done on the way the **Generic HASKELL** module system works. Also, a more formal specification on the **Generic HASKELL** library is needed.

From a theoretical standpoint, there is a gap between the fixed point approach to generic programming taken in [2] and [4] and the approach taken here. It is inherently not possible to define catamorphisms and anamorphisms in **Generic HASKELL**, whereas these functions play a central part in most related work on generic programming. Research is currently going on at Utrecht University on *views* which allow a data type to be viewed in different ways.

12.4. Conclusions

The techniques described in this thesis are a viable way of implementing generic programming. This has been demonstrated by the implementation of the **Generic HASKELL** compiler, which is based on a first prototype developed by Jan de Wit as part of this thesis. The **Generic HASKELL** library shows that a large number of generic functions can be defined in **Generic HASKELL**.

Bibliography

- [1] A. Alimarine and R. Plasmeijer. A generic programming extension for clean. In *Proceedings of the 13th International workshop on the Implementation of Functional Languages, IFL'01*, pages 257–278, 2001.
- [2] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In S. Doaitse Swierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999.
- [3] Ralf Hinze. *Generic Programs and Proofs*. 2000. Habilitationsschrift, Bonn University.
- [4] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.
- [5] Johan Jeuring. Generic programming for XML tools. In preparation.
- [6] Mark P. Jones. First-class polymorphism with type inference. In *Twenty Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97, Paris, France*, January 1997.
- [7] Simon Peyton Jones, John Hughes (editors), et al. Haskell 98 — A non-strict, purely functional language. Available from <http://haskell.org>, February 1999.
- [8] Simon Peyton Jones, John Hughes (editors), et al. Standard libraries for Haskell 98. Available from <http://haskell.org>, February 1999.
- [9] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.