The Generic Haskell User's Guide

Version 1.23 — Beryl release

Dave Clarke

Johan Jeuring

Andres Löh

institute of information and computing sciences, utrecht university

technical report UU-CS-2002-047

www.cs.uu.nl

Institute of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands
http://www.generic-haskell.org

The Generic H∀SKELL User's Guide

Version 1.23 — Beryl release

The **Generic H∀SKELL** Team

Dæv Clarke
Johan Jeuring
Andres Löh

info@generic-haskell.org

Contents

1 What is Generic H∀SKELL?

1.1 Generic programming

Software development often consists of designing datatypes around which functionality is added. Some functionality is datatype specific, whereas other functionality is defined on almost all datatypes in a way that depends only on the structure of the datatype. A function that works on many datatypes in this manner is called a *generic* (or polytypic) *function*. Examples of generic functionality include editing, pretty-printing or storing a value in a database, and comparing two values for equality.

Since datatypes often change and new datatypes are introduced, we have developed **Generic H∀SKELL** which supports generic definitions to save the programmer from (re)writing the instances of generic functions that would otherwise be requried. **Generic H∀SKELL** is based on recent work by Hinze [3], and extends the functional programming language Haskell [6] with, among other things, a construct for defining type-indexed values with kind-indexed types. These values can be specialised to all Haskell datatypes, facilitating wider application of generic programming than provided by earlier systems such as PolyP [5].

1.2 Generic H∀SKELL overview

Generic H∀SKELL extends Haskell with a number of features.

- *type-indexed values* are defined as a value indexed over the various Haskell type constructors (unit, primitive types, sums, products, and user-defined type constructors). In addition, we can also specify the behaviour of a type-indexed values for a specific constructor using *constructor cases*, and reuse one generic definition in another using *default cases*.
  The resulting type-indexed value can be specialised to any type.
- *kind-indexed types* are types indexed over kinds, defined by giving a case for both $*$ and $\kappa \to \kappa'$. Instances are obtained by applying the kind-indexed type to a kind.
- generic definitions can be used by applying them to a type or kind. This is called *generic application*. The result is a type or value, depending on which sort of generic definition is applied.
- *generic abstraction* enables generic definitions be defined by abstracting a type parameter (of a given kind).

- *type-indexed types* are types which are indexed over the type constructors. These can be used to give types to more involved generic values. The resulting type-indexed types can be specialised to any type.

2 Installation

2.1 System requirements

Generic H∀SKELL is written in Haskell 98. Minor parts require rank-2-polymorphism, as implemented, for example, by both GHC and Hugs. The package has been tested with GHC and comes with a `Makefile` suitable to build it using GHC. Although parts of the system are written using Utrecht University's attribute grammar system `ag` and Ralf Hinze's `frown :-(` parser generator, these tools are not required to build the compiler. The code generated by **Generic H∀SKELL** is Haskell 98 compliant, except that functions generated for higher kinded types require rank-n-polymorphism. (Fortunately higher-kinded types are not particularly common for most applications.)

Two kinds of distribution are available.

The *binary* distribution includes a `gh` compiler binary and can be used together with any Haskell system, as it translates **Generic H∀SKELL** input files into ordinary Haskell files. We currently provide binaries for Linux, Solaris, and Windows. In principle, we can provide binaries for any platform supported by GHC.

The *source* distribution includes the Haskell code for the `gh` compiler which has been generated from our compiler source using both `ag` and `frown :-(`. GHC is required to compile this distribution. Configuration files are provided.

At present, the compiler source (including both `ag` and `frown :-(` sources) may only be obtained by emailing `info@generic-haskell.org`.

2.2 Installing the binary distribution

Installation is simple. Instructions for Unix users are as follows. Other users must consult the appropriate `INSTALL` file.

```
./configure --prefix=install_path
```

`--prefix=install_path` is optional and defaults to `/usr/local/`.

Next run (GNU make is required):

```
make install
```

This will install the binary `gh` in the directory `${prefix}/bin`.

2.3 Building from source

Building from source requires exactly the same command sequence as for installing the binary distribution. The difference of course is the amount of work which is subsequently done.

Note that for the source distribution, the install path prefix defaults to the directory you unpacked the distribution to. To get the same behaviour as in the binary distribution, explicitly call `configure` as:

```
./configure -prefix=/usr/local
```

2.4 Running gh

The **Generic H∀SKELL** compiler is called `gh`. It has essentially two modes of operation.

- If you call `gh` without supplying a file to process, you will be asked for a file name. Specify an input file relative to your working directory, and the compiler will process the file and generate an output file with the same basename as the input file, but the extension `.hs`.
- Alternatively, input files can be specified on the command line.
  A typical invocation is:
  `path_to_gh/bin/gh your_file.ghs`

Options A number of command line options are also available:

```
Usage: gh [options...] files...
  -v      --verbose             (number of v's controls the verbosity)
  -m      --make                follow dependencies
  -V      --version, --release  show version info
  -h, -?  --help                show help
          --cut=N               cut computations after N iterations
  -C      --continue            continue after errors
  -L DIR  --library=DIR         add DIR to search path
```

The first level of verbosity (no `-v` flag) produces only error messages. The second level (`-v`) in addition provides some diagnostic information and warnings. The third level (`-vv`) in addition provides debugging information.

The `-m` (or `--make`) option attempts to chase dependencies and compile those which require compilation. There are unfortunately cases where this fails. This feature will be improved for future releases.

The option `--cut=N` stops the compilation after `N` iterators of the specialisation mechanism. If this limit is reached, compilation fails. The first level of verbosity (`-v`) can be used to report the number of iterations used.

The `-C` (or `--continue`) option forces the compiler to continue compilation even when an error is encountered. This can be used to generate more than one error message or to see the resulting generated code, but unfortunately results in the compiler running until it crashes.

Search path The **Generic H∀SKELL** compiler needs to find a number of files, in particular the **Generic H∀SKELL** prelude. There are three possibilities to make the location of the standard libraries known to the compiler:

- Set the environment variable `GH_HOME` to the directory you unpacked the **Generic H∀SKELL** distribution to.
- Set the environment variable `GH_LIBRARY_PATH` to the directory where the libraries are located (usually `GH_HOME/lib`).
- Pass the path where the libraries are located as an argument to the compiler, using the `-L` option. This option can also be used to add other directories to the search path.

2.5 General overview of compilation

The **Generic H∀SKELL** compiler compiles `.ghs` files and produces `.hs` files which can subsequently be compiled using a Haskell compiler. In addition, the compiler also produces `.ghi` interface files for compiled modules, which will be used in subsequent compilations to to avoid unnecessary recompilation.

2.6 Compiling and running the generated code

The **Generic H∀SKELL** compiler generates ordinary Haskell code which can be run or compiled using GHC, Hugs, or any other Haskell compiler. Ensure that you include the path to `GHPrelude.hs` (and other library files you might be using), which can be found in the `lib` subdirectory, in your compiler's search path.

3 Generic H∀SKELL: The Language

The **Generic H∀SKELL** compiler implements a number of extensions to Haskell. These are described briefly here. Further information can also be found by consulting the literature [1, 2, 3, for example] and in the library included in the distribution (in `lib/*.ghs`).

3.1 Special Parentheses

Kind-indexed and type-indexed definitions take a (single) kind or type argument which is surrounded by special parentheses. The parentheses $\{\!|\_|\!\}$ (i.e., `{[ ]}`) wrap a kind argument, whereas $\{\!|\_|\!\}$ (i.e., `{| |}`) wraps a type argument.

3.2 Kind-indexed types

Type-indexed values (may) possess kind-indexed types. Kind-indexed types are defined in **Generic H∀SKELL** using a new top-level declaration which has the following syntax:

$$\begin{array}{lcl} \textbf{type}\ \langle Conid\rangle\{\!|*|\!\}\ t1\ \ldots\ tn & = & \langle type\rangle \\ \textbf{type}\ \langle Conid\rangle\{\!|k \to l|\!\}\ t1\ \ldots\ tn & = & \langle type\rangle \end{array}$$

A case is defined for both kind $*$ and for higher kinds $k \to l$. To a certain degree the $k \to l$ case is predetermined, depending on the number of arguments [3]. This is exemplified by the $k \to l$ case in the following example.

Example The kind-indexed type for the generic *map* function is defined as:

$$\begin{array}{lcl} \textbf{type}\ Map\{\!|*|\!\}\ t1\ t2 & = & t1 \to t2 \\ \textbf{type}\ Map\{\!|k \to l|\!\}\ t1\ t2 & = & \\ \qquad \textbf{forall}\ u\ v\ .\ Map\{\!|k|\!\}\ u\ v \to Map\{\!|l|\!\}\ (t1\ u)\ (t2\ v) & & \end{array}$$

Note that both cases have the same number of arguments and an equal number of type variables introduced by the **forall**.

Sometimes we wish to thread a particular type variable through such a definition, for example, when using a monadic type.

$$\begin{array}{lcl} \textbf{type}\ MapM\{\!|*|\!\}\ t1\ t2\ m & = & t1 \to m\ t2 \\ \textbf{type}\ MapM\{\!|k \to l|\!\}\ t1\ t2\ m & = & \\ \qquad \textbf{forall}\ u\ v\ .\ MapM\{\!|k|\!\}\ u\ v\ m \to MapM\{\!|l|\!\}\ (t1\ u)\ (t2\ v)\ m & & \end{array}$$

3.3 Type-indexed values

Generic H∀SKELL introduces a new top-level declaration for type-indexed values. A type-indexed value is defined using the following syntax:

$$
\begin{array}{lcl}
\langle Varid\rangle \{\!|\ t :: k\ |\!\} & :: & \langle type\rangle \\
\langle Varid\rangle \{\!|\ \langle stype\rangle\ |\!\} \ldots & = & \ldots \\
\langle Varid\rangle \{\!|\ \langle stype\rangle\ |\!\} \ldots & = & \ldots \\
\qquad\qquad\qquad\qquad \vdots & &
\end{array}
$$

where

$$\langle stype\rangle ::= Unit \mid \texttt{:+:} \mid \texttt{:*:} \mid Fun \mid Con\ \langle var\rangle \mid Label\ \langle var\rangle \mid \langle tycon\rangle \mid \textbf{case}\ \langle qcon\rangle$$

Firstly, we must declare the type of the type-indexed value. This is generally an expression involving a kind- or type-indexed type. Then we provide its definition as a collection of cases indexed over the constructors ($\langle stype\rangle$).

Corresponding to each $\langle stype\rangle$ is a regular Haskell **data** or **type** declaration. It is important to know these so that the appropriate pattern can be employed when coding each case of a generic definition.

```
data Unit                    =  Unit
data Sum a b                 =  Inl a | Inr b
                     -- Sum corresponds to :+: in type indices
data Prod a b                =  a :*: b
                     -- Prod corresponds to :*: in type indices
type Fun                     =  (→)
data Con a                   =  Con a
data Label a                 =  Label a
```

These are defined in the file `GHPrelude.hs` which is automatically imported by the generated code.

If used as a type argument, type constructors *Con* and *Label* get an extra argument which can be bound to a variable and is of type *ConDescr* or *LabelDescr*, respectively.

Note: The types *Con* and *Label* do no longer contain the descriptors themselves. This was a redundancy in the Amber release that has lead to confusion and has therefore been changed. Old programs have to be adapted to the change.

The datatypes *ConDescr* and *LabelDescr* are given here. These can be used for manipulating constructors and labels.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{ConDescr} & = & \mathit{ConDescr}\{\ \mathit{conName} :: \mathit{String},\\
 & & \quad\ \mathit{conType} :: \mathit{String},\\
 & & \quad\ \mathit{conArity} :: \mathit{Int},\\
 & & \quad\ \mathit{conLabels} :: \mathit{Bool},\\
 & & \quad\ \mathit{conFixity} :: \mathit{Fixity}\ \}\\
\textbf{data}\ \mathit{Fixity} & = & \mathit{Nonfix}\\
 & \mid & \mathit{Infix}\{\ \mathit{prec} :: \mathit{Int}\ \}\\
 & \mid & \mathit{Infixl}\{\ \mathit{prec} :: \mathit{Int}\ \}\\
 & \mid & \mathit{Infixr}\{\ \mathit{prec} :: \mathit{Int}\ \}\\
\textbf{data}\ \mathit{LabelDescr} & = & \mathit{LabelDescr}\{\ \mathit{labelName} :: \mathit{String},\\
 & & \quad\ \mathit{labelType} :: \mathit{String},\\
 & & \quad\ \mathit{labelStrict} :: \mathit{Bool}\ \}
\end{array}
$$

Consult `GHPrelude.hs` for details of other auxiliary functions.

Naturally, none of these special identifiers should be used in the remainder of a program in a way that clashes with their use in generic definitions, following the usual scoping rules of Haskell.

Example The type-indexed value for the generic *map* function is defined as:

$$
\begin{array}{lllcl}
\mathit{map}\{\!|t :: k|\!\} & & & :: & \mathit{Map}\{\!|k|\!\}\ t\ t\\
\mathit{map}\{\!|\mathit{Unit}|\!\} & & \mathit{Unit} & = & \mathit{Unit}\\
\mathit{map}\{\!|\texttt{:+:}|\!\} & \mathit{mA}\ \mathit{mB} & (\mathit{Inl}\ a) & = & \mathit{Inl}\ (\mathit{mA}\ a)\\
\mathit{map}\{\!|\texttt{:+:}|\!\} & \mathit{mA}\ \mathit{mB} & (\mathit{Inr}\ b) & = & \mathit{Inr}\ (\mathit{mB}\ b)\\
\mathit{map}\{\!|\texttt{:*:}|\!\} & \mathit{mA}\ \mathit{mB} & (a\texttt{:*:}b) & = & \mathit{mA}\ a\texttt{:*:}\mathit{mB}\ b\\
\mathit{map}\{\!|\mathit{Con}\ c|\!\} & m & (\mathit{Con}\ b) & = & \mathit{Con}\ (m\ b)\\
\mathit{map}\{\!|\mathit{Label}\ l|\!\} & m & (\mathit{Label}\ b) & = & \mathit{Label}\ (m\ b)\\
\mathit{map}\{\!|(\rightarrow)|\!\} & & & = & \\
 & \multicolumn{3}{l}{\mathit{error}\ \texttt{"cannot␣map␣over␣function␣type"}}\\
\mathit{map}\{\!|\mathit{Int}|\!\} & & i & = & i\\
\mathit{map}\{\!|\mathit{Char}|\!\} & & c & = & c
\end{array}
$$

This function can also be implemented in a point-free style (see [2]).

Generics defined over $\langle \mathit{tycon} \rangle$ Type-indexed values (and later types) can be defined over (possibly user-defined) type constructors. This covers the case for *Int* and *Char*, as illustrated above. Additional cases such as the following are also possible (though in this case superfluous):

$$
\begin{array}{lcl}
\mathit{map}\{\!|\mathit{List}|\!\}\ m\ \mathit{Nil} & = & \mathit{Nil}\\
\mathit{map}\{\!|\mathit{List}|\!\}\ m\ (\mathit{Cons}\ a\ as) & = & \mathit{Cons}\ (m\ a)\ (\mathit{map}\{\!|\mathit{List}|\!\}\ m\ as)
\end{array}
$$

for the user defined type

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

Notice the call $\mathit{map}\{\!|\mathit{List}|\!\}\ m\ \mathit{as}$ on the right-hand side of this definition. This is required since *List* is a recursive type.

Constructor cases Often datatypes which have a large number of constructors require functions that behave in some uniform manner for most constructors, but in some specific way for certain other constructors. To write such functions **Generic HASKELL** allows cases for specific constructors to be written. Using these *constructor cases* a generic function can have special cases to deal with the constructors requiring special treatment.

The syntax of the case is given by

$$\textbf{case}\ \langle \mathit{qcon} \rangle$$

as illustrated in the following

$$\begin{array}{l} \mathit{freecollect}\{\!|\ \textbf{case}\ \mathit{Lambda}|\!\}\ (\mathit{Lambda}\ (v,t)\ e) \\ \qquad\qquad = \quad \mathit{filter}\ (\not\equiv v)\ (\mathit{freecollect}\{\!|\mathit{Expr}|\!\}\ e) \end{array}$$

The case $\mathit{freecollect}\{\!|\ \textbf{case}\ \mathit{Lambda}|\!\}$ will be applied only when the value of type *Expr* (from which *Lambda* is a constructor) encountered has the form $\mathit{Lambda}\ (v,t)\ e$. The case should be written to exploit this knowledge. Interestingly, when a constructor case produces a value, it need not produce a value with the same constructor, but only of the correct type.

The type of a constructor case is the same as the type from which the constructor comes. Thus, since *Lambda* is a constructor for the *Expr* datatype, the type of the right-hand side is what it would be for the *Expr* case.

Specific constructor cases of a generic function can be called, though we do not expect that this is particularly useful.

Default cases Default cases allow one generic definition to be defined by implicitly copying the lines from another, updating and adding cases where appropriate. This is particularly useful for defining functions which follow a specific traversal pattern.

Suppose we have a crush-like function which collects a list of values of type a from some datatype.

$$\begin{array}{lcl} \textbf{type}\ \mathit{Collect}\{\![*]\!\}\ t & = & t \to [\,a\,] \\ \textbf{type}\ \mathit{Collect}\{\![\kappa \to \nu]\!\}\ t & = & \textbf{forall}\ u\ .\ \mathit{Collect}\{\![\kappa]\!\}\ u \\ & & \qquad \to \mathit{Collect}\{\![\nu]\!\}\ (t\ u) \\ \mathit{collect}\{\!|t :: \kappa|\!\} & :: & \mathit{Collect}\{\![\kappa]\!\}\ t \\ \ldots & & \end{array}$$

We can adapt this function to collect values of type *Var*, to produced a function of the following more specific type

$$\begin{array}{lcl}
\textbf{type } \mathit{VarCollect}\{\!|*|\!\}\ t & = & t \to [\,\mathit{Var}\,] \\
\textbf{type } \mathit{VarCollect}\{\!|\kappa \to \nu|\!\}\ t & = & \textbf{forall } u\,.\ \mathit{VarCollect}\{\!|\kappa|\!\}\ u \\
 & & \qquad \to \mathit{VarCollect}\{\!|\nu|\!\}\ (t\ u),
\end{array}$$

by writing but a few lines:

$$\begin{array}{lcl}
\mathit{varcollect}\{\!|t :: \kappa|\!\} & :: & \mathit{VarCollect}\{\!|\kappa|\!\}\ t \\
\mathit{varcollect}\{\!|\mathit{Var}|\!\}\ v & = & [\,v\,] \\
\mathit{varcollect}\{\!|\texttt{:*:}|\!\}\ mA\ mB\ (a \texttt{:*:} b) & = & mA\ a \cup mB\ b \\
\mathit{varcollect}\{\!|c|\!\} & = & \mathit{collect}\{\!|c|\!\}\,.
\end{array}$$

The line "$\mathit{varcollect}\{\!|c|\!\} = \mathit{collect}\{\!|c|\!\}$" is the default case, which has the effect of copying the code from *collect* into the new generic function *varcollect*. The line for $\mathit{varcollect}\{\!|\mathit{Var}|\!\}$ specifies the desired additional functionality for type *Var*. The line for $\mathit{varcollect}\{\!|\texttt{:*:}|\!\}$ overrides the functionality for `:*:`, using union instead of concatenation to accumulate the results.

Both constructor and default cases are described elsewhere in more detail [1].

3.4 Generic application

A type-indexed value can be specialised to a value by applying it to a type. Generic application extends the syntax of expressions ($\langle aexp \rangle$) as follows:

$$\begin{array}{lcl}
\langle aexp \rangle & ::= & \ldots \\
 & \mid & \langle Varid \rangle\{\!|\langle type \rangle|\!\}
\end{array}$$

Similarly, a kind-indexed type can be specialised to a type by supplying the kind at which the definition is to be applied. The syntax of type expressions ($\langle gtycon \rangle$) is thus extended as follows:

$$\begin{array}{lcl}
\langle gtycon \rangle & ::= & \ldots \\
 & \mid & \langle Conid \rangle\{\!|\langle kind \rangle|\!\}
\end{array}$$

Example Given the datatype:

$$\textbf{data } \mathit{BinTree}\ a = \mathit{Empty} \mid \mathit{Node}\ a\ (\mathit{BinTree}\ a)\ (\mathit{BinTree}\ a)$$

The *map* function for *BinTree* is $\mathit{map}\{\!|\mathit{BinTree}|\!\}$. The type of $\mathit{map}\{\!|\mathit{BinTree}|\!\}$ is $\mathit{Map}\{\!|* \to *|\!\}\ \mathit{BinTree}\ \mathit{BinTree}$, which is $(a \to b) \to (\mathit{BinTree}\ a \to \mathit{BinTree}\ b)$.

3.5 Generic abstraction

A type variable (of fixed kind) can be abstracted generically from an expression using a kind of generic abstraction. Declarations take the following form:

$$\begin{array}{lcl} \langle Varid \rangle \; \{\!|\; t :: \langle kind \rangle \; |\!\} & :: & \langle type \rangle \\ \langle Varid \rangle \; \{\!|\; t \; |\!\} \ldots & = & \langle exp \rangle \end{array}$$

Here t is a type variable of the given kind, where $\langle kind \rangle$ ranges over grounded kinds (i.e., those without kind variables).

An example is the so-called categorical strength:

$$\begin{array}{lcl} strength\{|t :: * \to *|\} & :: & t\ a \to b \to t\ (a, b) \\ strength\{|t|\}\ ta\ b & = & map\{|t|\}\ (\lambda x \to (x, b))\ ta \end{array}$$

3.6 Type-indexed types

Type-indexed types [4] can be defined just as type-indexed values, except that the right-hand side of a definition is a constructor followed by a type. Thus the syntax consists of a collection of definitions, indexed over the type constructors, of the form:

$$\textbf{type}\ \langle Conid \rangle \{|\langle stype \rangle|\}\ tv1 \ldots tvn = \langle con \rangle\ \langle type \rangle$$

New constructors ($\langle con \rangle$) must be introduced for each case of such a definition — each case will be compiled into a **newtype** declaration.

A type-indexed type can be specialised to a type by supplying its type argument.

$$\begin{array}{lcl} \langle gtycon \rangle & ::= & \ldots \\ & \mid & \langle Conid \rangle \{|\langle type \rangle|\} \end{array}$$

Example The type-indexed type *FMap* is defined as follows:

$$\begin{array}{lcl} \textbf{type}\ FMap\{|Unit|\}\ v & = & FMU\ (Maybe\ v) \\ \textbf{type}\ FMap\{|:+:|\}\ fma\ fmb\ v & = & FMP\ (fma\ v, fmb\ v) \\ \textbf{type}\ FMap\{|:*:|\}\ fma\ fmb\ v & = & FMT\ (fma\ (fmb\ v)) \\ \textbf{type}\ FMap\{|Con|\}\ fm\ v & = & FMC\ (fm\ v) \\ \textbf{type}\ FMap\{|Label|\}\ fm\ v & = & FML\ (fm\ v) \end{array}$$

FMap can be used anywhere a type can by supplying *FMap* with its type parameter, for example in the following:

$$\begin{array}{lcl} \textbf{type}\ Lookup\{|*|\}\ t & = & \textbf{forall}\ v\ .\ FMap\{|t|\}\ v \to t \to Maybe\ v \\ \textbf{type}\ Lookup\{|k \to l|\}\ t & = & \textbf{forall}\ a\ .\ Lookup\{|k|\}\ a \to Lookup\{|l|\}\ (t\ a) \end{array}$$

The constructors introduced in the definition of *FMap* can be used in pattern matching:

$$
\begin{array}{llll}
lookup\{\!|\, t :: k \,|\!\} & & :: & Lookup\{\![k]\!\}\ t \\
lookup\{\!|\, Unit \,|\!\} & (FMU\ fm) \quad Unit & = & fm \\
lookup\{\!|\, \texttt{:+:} \,|\!\} & lA\ lB\ (FMP\ (fma, fmb))\ (Inl\ a) & = & lA\ fma\ a \\
lookup\{\!|\, \texttt{:+:} \,|\!\} & lA\ lB\ (FMP\ (fma, fmb))\ (Inr\ b) & = & lB\ fmb\ b \\
lookup\{\!|\, \texttt{:*:} \,|\!\} & lA\ lB\ (FMT\ fma)\ (a\texttt{:*:}b) & = & \mathbf{do}\ fmb \leftarrow lA\ fma\ a \\
 & & & \quad lB\ fmb\ b \\
lookup\{\!|\, Con\ d \,|\!\} & l \quad (FMC\ fm) \quad (Con\ b) & = & l\ fm\ b \\
lookup\{\!|\, Label\ d \,|\!\} & l \quad (FML\ fm) \quad (Label\ b) & = & l\ fm\ b
\end{array}
$$

Note: This feature is experimental and subject to revision.

3.7 Specialisation

Generic functions are specialised at compile time, thus no run-time representation of types is required. (There is however the cost of encoding and decoding types.) The compiler determines which specific types a generic function is used with, and then generates the set of specialised versions for that function in the output file.

Specialisations are always generated locally per module. Thus, a generic function which is defined in one module but used in many, results in some work being duplicated.

The compiler proceeds by collecting *specialisation requests* and *implications* from the source. The implications are then applied to the requests repeatedly, yielding new requests, until a fixpoint is reached.

In the presence of generic abstractions, where other generic functions may be called with an arbitrarily complex type argument, the specialisation process may fail to terminate. The compiler can be forced to quit after a certain number of iterations with the `--cut` command line option.

3.8 Generated function naming

The **Generic H∀SKELL** programmer must be aware that the generated Haskell code is poluted with additional names corresponding to instances of generic functions. These may clash with a programmer's own function names. Fortuately, this is highly unlikely as the generated names are rather complicated, encoding details such as module and type names. Unfortunately, this obfuscation makes it difficult to directly interface ordinary Haskell code with the code generated by the **Generic H∀SKELL** compiler. We offer a tip to the adventurous who wish to do such a thing. If you wish to use a generic function such as $map\{\!|\, List \,|\!\}$ in ordinary haskell code, add a line such as

$$mapList = map\{\!|\, List \,|\!\}$$

to the appropriate **Generic H∀SKELL** file, and then use the function *mapList* in your Haskell code.

3.9 Module system

The module system of **Generic H∀SKELL** mirrors the behaviour of Haskell's module system, as far as the Haskell language is concerned. Additionally, generic entities (i.e. kind-indexed types, type-indexed values, and type-indexed types) may appear in export and import lists. If no export or import list is given, then all generic entities are exported or imported, respectively. If a generic entity appears in a list, then all of its cases are exported or imported. It is not possible to export only some cases of a type-indexed value, or to limit the constructors visible for a type-indexed type.

It is recommended that the kind-indexed type of a type-indexed value is also exported. Forgetting to do so may result in unexpected behaviour.

In contrast to the previous (Amber) release, qualified names work everywhere. It is possible to import and use generic entities qualified. Defining a type-indexed value or type-indexed type across modules is not possible. However, you can achieve a similar effect by importing a generic function qualified and redefining a new function with the same name by means of a default case.

3.10 Haskell compatibility

Generic H∀SKELL parses all Haskell programs, except in the following instances:

- The token **forall** is an additional keyword in **Generic H∀SKELL**. As this is already the case in the extensions provided by many Haskell implementations, it should hopefully not cause too much trouble.
- The special parentheses for type and kind arguments, i.e. {|, |}, {[|, |]}, are all handled as a single token. Unfortunately, some pieces of regular Haskell code can trick the lexer and result in parse errors. For example, in

  $$\mathbf{do}\ \{[x] \leftarrow action; return\ x\}$$

  the initial {[is treated as a single token {[| rather than the two tokens { and [which an Haskell programmer would expect. In other instances, sequences such as +|} are considered as the operator +| followed by a }, since | may occur in operators, whereas {|+ is considered as the token {| followed by +.
  The required fix in both cases is to insert a space in the appropriate place, for example, by writing instead $\mathbf{do}\ \{_{\sqcup}\,[x] \leftarrow action; return\ x\}$.

4 Library

4.1 Introduction

Provided with the **Generic H∀SKELL** system is a library of useful generic functions. These are summarised below; for the details, consult the library itself (in subdirectory `lib`). We give the types of the generic functions for kind $*$ and $* \to *$, and usually a short description.

Naming conventions When generic functions defined in the **Generic H∀SKELL** library have an equivalent in the Haskell Prelude or libraries, the name of the generic function is prefixed with a '`g`'.

4.2 Module `Bounds`

$$\begin{array}{lcl} \mathit{gminBound}, \mathit{gmaxBound}\{\!|t :: *|\!\} & :: & t \\ \mathit{gminBound}, \mathit{gmaxBound}\{\!|t :: * \to *|\!\} & :: & a \to t\ a \end{array}$$

These are slight generalisations of the *minBound* and *maxBound* members of the *Bounded* class. They have the property that for all types t of kind $*$:

$$\forall\ a :: t\ .\ \mathit{gminBound}\{\!|t|\!\} \leqslant a \leqslant \mathit{gmaxBound}\{\!|t|\!\}$$

However, 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 [6, Appendix D]).

4.3 Module `Collect`

The functions in this module collect information about types and values of these types.

$$\begin{array}{lcl} \mathit{constructorOf}\{\!|t :: *|\!\} & :: & t \to \mathit{ConDescr} \\ \mathit{constructorOf}\{\!|t :: * \to *|\!\} & :: & (a \to \mathit{ConDescr}) \to t\ a \to \mathit{ConDescr} \end{array}$$

constructorOf returns a description of the topmost constructor in a value.

$$\begin{array}{lcl} \mathit{constructors}\{\!|t :: *|\!\} & :: & [\,\mathit{ConDescr}\,] \\ \mathit{constructors}\{\!|t :: * \to *|\!\} & :: & [\,\mathit{ConDescr}\,] \to [\,\mathit{ConDescr}\,] \end{array}$$

constructors returns a list of descriptions of all topmost constructors used in a datatype.

$$\begin{array}{lcl} \mathit{labelsOf}\{\!|t :: *|\!\} & :: & t \to [\mathit{LabelDescr}] \\ \mathit{labelsOf}\{\!|t :: * \to *|\!\} & :: & (a \to \mathit{LabelDescr}) \to t\ a \to \mathit{LabelDescr} \end{array}$$

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

$$\begin{array}{lcl} \mathit{constructorsAndLabels}\{\!|t :: *|\!\} & :: & [(\mathit{ConDescr}, [\mathit{LabelDescr}])] \\ \mathit{constructorsAndLabels}\{\!|t :: * \to *|\!\} & :: & [(\mathit{ConDescr}, [\mathit{LabelDescr}])] \\ & & \quad \to [(\mathit{ConDescr}, [\mathit{LabelDescr}])] \end{array}$$

constructorsAndLabels combines the above information: it returns a list of all constructors, paired with the labels present in the given type constructor.
Again, consult `GHPrelude.hs` for details of *ConDescr* and *LabelDescr*.

4.4 Module `Compare`

$$\begin{array}{lcl} \mathit{gcompare}\{\!|t :: *|\!\} & :: & t \to t \to \mathit{Ordering} \\ \mathit{gcompare}\{\!|t :: * \to *|\!\} & :: & (a \to b \to \mathit{Ordering}) \to t\ a \to t\ b \to \mathit{Ordering} \end{array}$$

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

4.5 Module `Eq`

$$\begin{array}{lcl} \mathit{eq}\{\!|t :: *|\!\} & :: & t \to t \to \mathit{Bool} \\ \mathit{eq}\{\!|t :: * \to *|\!\} & :: & (a \to b \to \mathit{Bool}) \to t\ a \to t\ b \to \mathit{Bool} \end{array}$$

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

4.6 Module `Map`

$$\begin{array}{lcl} \mathit{gmap}\{\!|t :: *|\!\} & :: & t \to t \\ \mathit{gmap}\{\!|t :: * \to *|\!\} & :: & (a \to b) \to t\ a \to t\ b \end{array}$$

gmap is the generic version of *fmap* in *Functor* class.

4.7 Module `MapM`

$$\begin{array}{lcl} \mathit{mapMl}, \mathit{mapMr}\{\!|t :: *|\!\} & :: & (\mathit{Functor}\ m, \mathit{Monad}\ m) \Rightarrow t \to m\ t \\ \mathit{mapMl}, \mathit{mapMr}\{\!|t :: * \to *|\!\} & :: & (\mathit{Functor}\ m, \mathit{Monad}\ m) \Rightarrow (a \to m\ b) \to t\ a \to m\ (t\ b) \end{array}$$

These are the generic versions of the monadic map *mapM* in the Prelude. *mapMl* traverses a data structure from left to right (just like *mapM*) while *mapMr* traverses from right to left. The *Monad* in the context should also be an instance of class *Functor*, but that is usually not problematic.

4.8 Module `ReadShow`

$$
\begin{array}{lcl}
gshowsPrec\{|t :: *|\} & :: & Bool \to Int \to t \to ShowS \\
gshowsPrec\{|t :: * \to *|\} & :: & (Bool \to Int \to a \to ShowS) \to \\
 & & \quad Bool \to Int \to t\ a \to ShowS \\
 & & \\
greadsPrec\{|t :: *|\} & :: & Bool \to Int \to ReadS\ t \\
greadsPrec\{|t :: * \to *|\} & :: & (Bool \to Int \to ReadS\ a) \to \\
 & & \quad Bool \to Int \to ReadS\ (t\ a)
\end{array}
$$

The generic versions of *show* and *read* (in classes *Show* and *Read*).

The extra argument of type *Bool* is used internally to specify whether field labels are to be printed (and separated by commas). It should usually be *False*.

Since calling these functions is a bit cumbersome, the following specialisations are provided:

$$
\begin{array}{lcl}
gshow\{|t :: *|\} & :: & t \to String \\
gshow1\{|t :: * \to *|\} & :: & Show\ a \Rightarrow t\ a \to String \\
gread\{|t :: *|\} & :: & String \to t \\
gread1\{|t :: * \to *|\} & :: & Read\ a \Rightarrow String \to t\ a
\end{array}
$$

4.9 Module `Reduce`

$$
\begin{array}{lcl}
rreduce\{|t :: *|\} & :: & t \to b \to b \\
rreduce\{|t :: * \to *|\} & :: & (a \to b \to b) \to t\ a \to b \to b \\
lreduce\{|t :: *|\} & :: & b \to t \to b \\
lreduce\{|t :: * \to *|\} & :: & (b \to a \to b) \to b \to t\ a \to b
\end{array}
$$

rreduce is a generic version of *foldr* (note the reversed order of the last two arguments!), while *lreduce* is a generic *foldl*. See [2, section 5.4].

$$
\begin{array}{lcl}
crush\{|t :: * \to *|\} & :: & (a \to a \to a) \to a \to t\ a \to a
\end{array}
$$

crush is an instance of *lreduce* with a slightly more familiar type.

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

$$
\begin{array}{lcl}
gsum, gproduct\{|t :: * \to *|\} & :: & Num\ a \Rightarrow t\ a \to a \\
gand, gor\{|t :: * \to *|\} & :: & t\ Bool \to a \\
flatten\{|t :: * \to *|\} & :: & t\ a \to [a] \\
count\{|t :: * \to *|\} & :: & t\ a \to Int \\
comp\{|t :: * \to *|\} & :: & t\ (a \to a) \to (a \to a) \\
gconcat\{|t :: * \to *|\} & :: & t\ [a] \to [a] \\
gall, gany\{|t :: * \to *|\} & :: & (a \to Bool) \to t\ a \to Bool \\
gelem\{|t :: * \to *|\} & :: & Eq\ a \Rightarrow a \to t\ a \to Bool
\end{array}
$$

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

4.10 Module `Table`

The module `Table` provides a type-indexed type and functions for building memo tables of functions. See [2, section 5.6] and the code in `lib/Table.ghs`.

4.11 Module `ZipWith`

$$\begin{array}{lll} gzipWith\{\!|t :: *|\!\} & :: & (t, t) \to Maybe\ t \\ gzipWith\{\!|t :: * \to *|\!\} & :: & ((a, b) \to Maybe\ c) \to (t\ a, t\ b) \to Maybe\ (t\ c) \end{array}$$

A generic version of *zipWith*, except that it returns *Nothing* when the two data structures do not have the same shape.

$$\begin{array}{lll} gunzipWith\{\!|t :: *|\!\} & :: & t \to (t, t) \\ gunzipWith\{\!|t :: * \to *|\!\} & :: & (a \to (b, c)) \to t\ a \to (t\ b, t\ c) \end{array}$$

gunzipWith is a generic version of *unzip*.

$$\begin{array}{lll} gzip\{\!|t :: * \to *|\!\} & :: & t\ a \to t\ b \to Maybe\ (t\ (a, b)) \\ gunzip\{\!|t :: * \to *|\!\} & :: & t\ (a, b) \to (t\ a, t\ b) \end{array}$$

These functions are more or less direct generalisations of *zip* and *unzip* respectively, defined as instances of *gzipWith* and *gunzipWith*.

5 Future Work

In the future, we plan to continue our work on the compiler. Among the many possible extensions and improvements, we are initially considering:

- support for POPL-style definitions
- adding a type checker
- a view mechanism (i.e. implicit maps between data types); better support for fixpoints
- improved support for type-indexed data types
- ...

As we have not yet decided how the next major release of the **Generic HA∀SKELL** compiler will look, these topics are subject to change. Any input and feedback is most welcome!

6 Meta-information

6.1 Contact

The Generic H∀SKELL Project For information regarding the **Generic H∀SKELL** project send email to `info@generic-haskell.org`.

Mailing List A low volume mailing list exists. Currently it serves as a place for distributing information relevant to **Generic H∀SKELL** and for announcing our project meetings. This is the appropriate forum for general language discussions and whatnot. The address is `generic-haskell@generic-haskell.org`. To subscribe to the mailing list send an email to `listadm@generic-haskell.org`.

Bug Reports Bugs can be reported to `bugs@generic-haskell.org`.

6.2 Caveats

The **Generic H∀SKELL** compiler is a research prototype. Many of its features, especially the more experimental ones, may change as we gain more experience and understanding.

It should be noted that the compiler does not perform type checking of the **Generic H∀SKELL** source language. Thus type errors in **Generic H∀SKELL** source will often be discovered only when the generated Haskell source is compiled.

6.3 Known bugs and limitations

1. The constructor descriptors for user-defined data types that have infix constructors with non-default fixity will be generated incorrectly with the default fixity.
2. Usage of the keyword **forall** in types is a bit tricky. There are places where it is needed, and others where it may cause strange errors. This will be clarified in the future. Let the examples guide you for now.
3. A type-indexed data type which is specialised to the same type in two separate modules results in types which should be the same, but are treated differently by Haskell.

4. A *datatype* (or **newtype**) which has a parameter of higher kinded type that does not appear in the right hand side of the definition produces a bug in the Haskell code generated by `gh`. The datatype Y below exhibits the behaviour.

   $$\textbf{data}\ X\ a = X\ a$$
   $$\textbf{data}\ Y\ b = Y\ (X\ (Y\ X))$$

   Parameter b has kind $* \to *$, but it does not occur in $Y\ (X\ (Y\ X))$. We do not imagine that such types are very common, so the bug will remain for now.

5. This list is incomplete.

6.4 Change log

Beryl (1.23) Syntax for using *Con* and *Label* in generic functions has slightly changed (see p. 10 for details). Added constructor and default cases. Improved support for the module system. Revamped specialisation mechanism — it is now demand driven and generates less code. Numerous bug fixes.

Amber (0.99) The first release.

6.5 Contributors

Ralf Hinze and Jan de Wit contributed significantly to earlier versions of **Generic H∀SKELL**.

6.6 Acknowledgements

Thanks to Ralf Hinze for `frown :-(`, to Arthur Baars and Doaitse Swiestra for `ag`, and to Simon Marlow and Sven Panne for the original Happy Haskell grammar.

6.7 Copyright information

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Bibliography

[1] Dave Clarke and Andres Löh. Generic Haskell, Specifically. In *IFIP 2.1 Working Conference on Generic Programming*, July 2002.

[2] Ralf Hinze. *Generic Programs and Proofs*. 2000. Habilitationsschrift, Bonn University. Available from `http://www.informatik.uni-bonn.de/~ralf/`.

[3] Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and José Nuno Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *LNCS*, pages 2–27. Springer-Verlag, 2000.

[4] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *Proceedings of the 6th Conference on Mathematics of Program Construction (MPC 2002)*. Springer Verlag, July 2002.

[5] 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.

[6] Simon Peyton Jones, John Hughes (editors), et al. Haskell 98 — A non-strict, purely functional language. Available from `http://haskell.org`, February 1999.