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

The **Generic H∀SKELL** Team

Dæv Clarke
Ralf Hinze
Johan Jeuring
Andres Löh
Jan de Wit

info@generic-haskell.org

November 1, 2001

Version 0.99 (Amber)

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 such a way that it depends only on the structure of the datatype. A function that works on many datatypes in this way is called a *generic* (or polytypic) *function*. Examples of generic functionality include storing a value in a database, editing a value, comparing two values for equality, pretty-printing a value.

Since datatypes often change and new datatypes are introduced, we have developed **Generic H∀SKELL**, an extension of the functional programming language Haskell [5] that supports generic definitions, to save the programmer from (re)writing instances of generic functions. **Generic H∀SKELL** extends Haskell with, among other things, a construct for defining type-indexed values with kind-indexed types, based on recent work by Hinze [2]. These values can be specialised to all Haskell datatypes, facilitating wider application of generic programming than provided by earlier systems such as PolyP [4].

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). 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* allows generic definitions to take a type parameter. Such a definition can apply only to types of a single 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 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 the Utrecht University's attribute grammar system `ag` and 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 particular common for most applications.)

Two kinds of distribution are available: Binary and Source.

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 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. Windows users must consult the `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 result in the binary `gh` being installed in the directory `${prefix}/bin`.

2.3 Building from source

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

2.4 Running `gh`

The **Generic H∀SKELL** compiler is called `gh`. There are essentially two modes for running it.

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 number of command line options are also available:

```
Usage: gh [options...] files...
  -v      --verbose             (number of v's controls the verbosity)
  -V      --version, --release  show version info
  -h, -?  --help                show help
  -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.

2.5 Compiling and running the generated code

The **Generic H∀SKELL** compiler generates ordinary Haskell code. This can be run using GHC, Hugs, or any other Haskell compiler. Ensure that you include the path to `GHPrelude.hs`, 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 be found by consulting the literature [2, 1, for example] and the examples included in the distribution (in `examples/*.ghs`).

3.1 Special Parentheses

Kind-indexed and type-indexed definitions take a (single) kind or type argument which is wrapped by special parentheses. The parentheses $\{\!|\sqcup|\!\}$ (i.e., `{[ ]}`) wraps a kind argument, whereas $\{\!|\sqcup|\!\}$ (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 [2]. 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:

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

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

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.

$$\langle Varid\rangle \ \{\!|\ t :: k\ |\!\}\ ::\ \langle type\rangle$$
$$\langle Varid\rangle \ \{\!|\ \langle stype\rangle\ |\!\} \ldots = \ldots$$

where

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

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 which is 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 in the appropriate 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 ConDescr a
data Label a              =  Label LabelDescr a
```

These are defined in the file `GHPrelude.hs` which is automatically imported by the generated code. Consult `GHPrelude.hs` for details of *ConDescr* and *LabelDescr* and for other auxiliary functions.

Naturally, these identifiers should not 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:

```
map{|t :: k|}                 ::  Map{|[k]|} t t
map{|Unit|} Unit              =   Unit
map{|:+:|} mA mB (Inl a)      =   Inl (mA a)
map{|:+:|} mA mB (Inr b)      =   Inr (mB b)
map{|:*:|} mA mB (a :*: b)    =   mA a :*: mB b
map{|Con c|} m (Con d b)      =   Con d (m b)
```

$$
\begin{array}{lcl}
map\{\!|Label\ l|\!\}\ m\ (Label\ l\ b) & = & Label\ l\ (m\ b) \\
map\{\!|(\rightarrow)|\!\} & = & error\ \texttt{"cannot␣map␣over␣function␣type"} \\
map\{\!|Int|\!\}\ i & = & i \\
map\{\!|Char|\!\}\ c & = & c
\end{array}
$$

This function can also be implemented in a point-free style (see `examples/Map.ghs`).

Generics defined over $\langle tycon \rangle$ 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):

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

for the user defined type

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

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

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:

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

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

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 [3] 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.

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

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.

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

Note: This feature is experimental and subject to revision.

3.7 Generated function naming

The **Generic HASKELL** programmer must (unfortunately) be aware of the names which the compiler uses in the generated Haskell source in order to avoid name clashes.

For a type-indexed value with with name *poly*, the name of the code generated for this function for type *Type* is *polyType*. This is the name of the function $poly\{\!|Type|\!\}$. Thus *map* for *BinTree* (or $map\{\!|BinTree|\!\}$) is called *mapBinTree*. There are a number of exceptions. The suffixes for type constructors [ ], (,), (, ,) and (→) are *LIST*, *TUPLE2*, *TUPLE3*, *Fun*, respectively. Thus the generated *map* functions for these types are *mapLIST*, *mapTUPLE2*, and so forth.

With this in mind, a programmer may link ordinary Haskell code with code generated by the **Generic HASKELL** compiler by using the names just described.

3.8 Haskell compatibility

Generic HASKELL parses all Haskell programs, except when one of the following problems occur:

- The token **forall** is an additional keyword in **Generic HASKELL**. 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`

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

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

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

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 [5, Appendix D]). For kinds other than $*$, the first parameters are not used: they can be $\bot$.

4.3 Module `Collect`

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

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

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

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

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

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

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

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

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

For kinds other than $*$, the first parameters are not used. Again, consult `GHPrelude.hs` for details of *ConDescr* and *LabelDescr*.

4.4 Module `Compare`

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

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

4.5 Module `Eq`

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

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

4.6 Module `Map`

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

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

4.7 Module `MapM`

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

These are the generic versions of the monadic map *mapM* in the Prelude. *mapMl* traverses a datastructure from left to right (just like *mapM*) while *mapMr* does this from right to left. The *Monad* in the context should also be an instance of class *Functor*, but that's 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, 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 [1, 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:

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

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

4.10 Module `Table`

See [1, section 5.6] and the code in `lib/Table.ghs`.

4.11 Module `ZipWith`

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

A generic version of *zipWith*, with the difference that in case the structures don't match, *Nothing* is returned.

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

gunzipWith is a generic version of *unzip*.

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

These functions are more direct generalizations of *zip* and *unzip* respectively. They are 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
- full support for modules
- 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
- XML, generic traversals, ...

As we have not yet decided what the next major release of the **Generic HA∀SKELL** compiler will look like, these topics are subject to change. Any input and feedback is 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 `info@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 only be discovered when the generated Haskell source is compiled.

6.3 Known bugs and limitations

1. Qualified names are not handled correctly in many parts of the compiler. If you want to have reliable behaviour, then avoid using any qualified names in your **Generic H∀SKELL** source files.
2. Export/import lists for modules are ignored by the compiler, which will lead to unwanted results if used for **Generic H∀SKELL** source files.
3. Imported types are not specialised for imported generic definitions. (This is not really a bug or a limitation, but rather prevents a generic function from being specialised to a specific type twice.)

4. The constructor descriptors for user-defined data types that have infix constructors with non-default fixity will be generated incorrectly with the default fixity.
5. 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.
6. We have not yet decided how to deal with generic definitions which are declared across module boundaries. Any behaviour that these may give is incidental and should not be considered as definitive.
7. Type-indexed data types do not work well with modules. All specialisations have to be generated in the module in which they are defined.
8. 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.

   data $X\ a = X\ a$
   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.

6.4 Change Log

Amber (0.99) Everything is new. This is the first release.

6.5 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.

6.6 Copyright information

Bibliography

[1] Ralf Hinze. *Generic Programs and Proofs*. 2000. Habilitationsschrift, Bonn University.

[2] 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.

[3] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed datatypes. Unpublished, 2001. `http://www.cs.uu.nl/~johanj/publications/tidata.ps`.

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