The Generic Haskell user's guide, Version 1.80 - Emerald release

Andres Löh

Johan Jeuring

Thomas van Noort

Alexey Rodriguez

Dave Clarke

Ralf Hinze

Jan de Wit

2

EMER∀LD

The Generic H∀SKELL User's Guide

Version 1.80 (Emerald)

The Generic H∀SKELL Team

Andres Löh
Johan Jeuring
Thomas van Noort
Alexey Rodriguez

Dave Clarke
Ralf Hinze
Jan de Wit

info@generic-haskell.org

April 11, 2008

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, an extension of the functional programming language Haskell [11] that supports generic definitions, to save the programmer from (re)writing instances of generic functions. The original design of Generic H∀SKELL is based on work by Ralf Hinze [3]. The current release is based on recent work by Dave Clarke, Johan Jeuring and Andres Löh [10, 9]. It extends Haskell with, among other things, a construct for defining type-indexed functions. These functions can be specialised to all Haskell datatypes, facilitating wider application of generic programming than provided by earlier systems such as PolyP [8].

1.2 Generic H∀SKELL overview

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

- *type-indexed functions* are defined as functions 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 function for a specific constructor using *constructor cases*, and reuse one generic definition in another using *default cases*.

  The resulting type-indexed function can be specialised to any type.

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

- generic definitions can be used by applying them to a type. This is called *generic application*. The result is a type or a function, depending on which sort of generic definition is applied.

- *generic abstraction* enables generic definitions to be defined by abstracting a type parameter (of a given kind).

2 Installation

2.1 System requirements

Generic H∀SKELL is written in Haskell. The package has been tested with GHC and comes with a `Makefile` suitable to build it using GHC. Parts of the system are written using Utrecht University's attribute grammar system `ag`, Ralf Hinze's `frown :-(` parser generator, and the DrIFT tool, but neither of these tools is required to build the compiler, because the generated Haskell sources are included in the distribution. The code that is generated by Generic H∀SKELL is such that it compiles using GHC, with extensions enabled. The generated code makes use of rank-n polymorphism and of explicit kind-annotations for data types – otherwise, it is Haskell 98 compliant.

Two kinds of distribution are available.

The *binary* distribution includes a `gh` compiler binary. The compiler translates Generic H∀SKELL input files into Haskell files, thus GHC is still required. We currently provide binaries for Linux and Mac OS X. 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`, `frown :-(`, and DrIFT. GHC is required to compile this distribution. Configuration files are provided.

Current development versions and snapshots can be obtained by accessing the Generic H∀SKELL Subversion repository at

> https://svn.cs.uu.nl:12443/repoman/info/Generic-Haskell.

2.2 Installing the binary distribution (Linux, Mac OS X)

Installation is also explained in the `INSTALL` file included in the distribution.

1. Unpack the tarball to a directory of your choice and change to this directory.

2. Issue the following commands:

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

   The flag `--prefix=install_path` is optional and defaults to `/usr/local/`. It can be used to select the place in the filesystem where the files specific to Generic H∀SKELL will be installed.

3. Next run (GNU make is required):

   ```
   make install
   make package
   ```

The first command will move all the files to their final locations. In particular, the binary `gh` will be placed in the directory `${prefix}/bin`. The second command will register the package `generic-haskell` – containing the Generic H∀SKELL libraries – with GHC.

Note that `make install` requires that you have write permissions to the installation directories, and `make package` requires that you have write permission to the global GHC package configuration file.

Alternatively, you can install Generic H∀SKELL for usage "in-place", using the command

```
make in-place
```

4. If you have chosen a global installation, you can remove the distribution directory now. For an in-place installation, it is still required.

2.3 Building from source

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

At the moment, the Windows platform is not supported. However, previous versions of Generic Haskell could be compiled under Cygwin, so given some effort it should be possible to produce working binaries.

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`. It will also generate an interface file with the extension `.ghi`.

- Alternatively, input files can be specified on the command line.

  A typical invocation is:

  ```
  gh your_file.ghs
  ```

  Again, the compiler will produce an `.hs` and a `.ghi` file as result.

2.5 Command line flags

A number of command line flags are available:

```
Usage: gh [options...] files...
  -v      --verbose             (number of v's controls the verbosity)
  -m      --make                compile dependent files first
```

```
    -V       --version, --release  show version info
    -h, -?   --help                show help
    -C       --continue            continue after errors
    -L DIR   --library=DIR         add DIR to search path
             --odir=DIR            place output files in DIR
             --print-searchpath    print search path
```

Verbosity (-v) The amount of output generated by the compiler can be selected by providing a number of `-v` flags. The default level of verbosity (no `-v` flag) produces only error messages. The second level (`-v`) in addition provides some diagnostic information and warnings. The third and fourth level (`-vv` and `-vvv`) in addition provide a significant amount of debugging information.

Track module dependencies (-m) The `-m` (or `--make`) option attempts to chase dependencies and compile those which require compilation. There are unfortunately cases where this fails.

Continue after errors (-C) 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 may result in the compiler running until it crashes.

Search path (-L) 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.

During installation, `GH_HOME` is preset to the installation directory, and the `gh` executable is in fact a wrapper script first setting the `GH_HOME` variable and then calling the real binary called `gh-bin`. If `GH_HOME` is set by the user, this information overrides the preset directory.

For debugging purposes, the search path used by Generic H∀SKELL can be queried using the `--print-searchpath` flag.

2.6 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 avoid unnecessary recompilation.

2.7 Compiling and running the generated code

The Generic H∀SKELL compiler generates ordinary Haskell code which can be run or compiled using GHC. The Generic H∀SKELL distribution includes wrapper scripts called `gh-ghc` and `gh-ghci`, for calling GHC and GHCi, respectively. These wrappers automatically pass some options appropriate for running programs generated by Generic H∀SKELL, such as making the libraries accessible.

2.8 Testing the compiler

The release provides several tests for the Generic H∀SKELL compiler, which can be found in the `tests` directory. The tests can be run from this directory using the command

```
../bin/testGH
```

Unfortunately, there are some tests which fail, as mentioned in Section 6.3.

3 Generic H∀SKELL: The Language

The Generic H∀SKELL compiler implements a number of extensions to Haskell. These are described briefly here, but this chapter is not intended as an introduction to generic programming. Further information can be found by consulting the literature [9, 5, 6, 2, for example]. In particular, Exploring Generic Haskell (EGH) [9] explains the Generic H∀SKELL language in much more detail than can be covered here. Most of the syntax used in EGH can be used in the Generic H∀SKELL compiler. Where there are significant differences, we point them out in this guide.

Furthermore, the program files included in the distribution (the `.ghs` files in the subdirectories `lib/`, `examples/`, and `tests/`) contain several examples of generic programming.

3.1 Special Parentheses

Type-indexed definitions take a type argument which is surrounded by special parentheses. The parentheses {⊔} (i.e., {| |}) wrap such a type argument. Although no longer required, we still support the parentheses {⊔} (i.e., {[ ]}), wrapping a kind argument of a kind-indexed type.

3.2 Type-indexed functions

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

$\langle varid \rangle$ {|$\langle gtyvars \rangle$|} :: $\langle gtypesig \rangle$
$\langle varid \rangle$ {|$\langle typat \rangle$|} ... = ...
$\langle varid \rangle$ {|$\langle typat \rangle$|} ... = ...
⋮

A type-indexed function consists of a type signature followed by one or more cases that define the function by matching on a type argument. Let us first look at these cases.

Type patterns   A type pattern {|$\langle typat \rangle$|} usually is the name of a type constructor, (fully) applied to distinct type variables, so that the resulting pattern is a type expression of kind ∗. For example,

Unit
[a]
Either a b
$a \rightarrow b$

10

are all valid type patterns, whereas

```
Either a    -- missing the second type variable
a → a       -- no distinct variables
Int → a     -- (→) is not applied to only variables
```

are all illegal.

Type patterns may also consist only of the name of a type constructor. Thus,

```
Either
Sum
(→)
```

are also valid type patterns. Such *named types* correspond to the style of generic programming that was supported by Generic H∀SKELL prior to the Coral release. If named types of a kind other than ∗ are used as type patterns, then the case of the function may take additional parameters according to the dependencies of the function. Generic H∀SKELL now also supports *dependency-style* definitions, which make use of the type patterns with type variables.

Structural representation types   Type-indexed functions become generic if they are defined for some or all of the structural representation types, which are used internally by Generic H∀SKELL to represent Haskell datatypes. These types are

```
data Unit      = Unit
data Sum a b = Inl a | Inr b
     -- Sum can be written as :+: in type indices
data Prod a b = a:*:b
     -- Prod can be written as :*: in type indices
data Con a     = Con a
data Label a  = Label a
```

These types are defined in the module `GH.Prelude`, which is automatically imported in each Generic H∀SKELL program. The types Con and Label are used to represent constructors and record field labels, respectively. When used as a type pattern, they get an additional argument which is bound to a *value* (not a type) of type ConDescr or LabelDescr.

The correct dependency-style type patterns for Con and Label are thus:

```
Con d a
Label d a
```

and as named types, they receive the additional argument as well:

```
Con d
Label d
```

The datatypes ConDescr and LabelDescr are given here. These can be used for querying information about constructors and labels.

```
data ConDescr   = ConDescr { conName     :: String,
                             conType     :: String,
                             conArity    :: Int,
                             conPosition :: (Int, Int),
                             conLabels   :: Bool,
                             conFixity   :: Fixity }
data Fixity      = Nonfix
                 | Infix  { prec :: Int }
                 | Infixl { prec :: Int }
                 | Infixr { prec :: Int }
data LabelDescr = LabelDescr { labelName   :: String,
                              labelType   :: String,
                              labelStrict :: Bool }
```

Naturally, none of these predefined 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 A type-indexed function that checks whether two given values are structurally equal can be defined as follows:

```
eq {|a :: *|}                              :: a → a → Bool
eq {|Unit|}     Unit       Unit       = True
eq {|Sum a b|} (Inl x)    (Inl y)    = eq {|a|} x y
eq {|Sum a b|} (Inr x)    (Inr y)    = eq {|b|} x y
eq {|Sum a b|}  _          _         = False
eq {|Prod a b|} (x₁ :*: x₂) (y₁ :*: y₂) = eq {|a|} x₁ y₁ ∧ eq {|b|} x₂ y₂
eq {|Int|}       m          n         = m == n    -- Haskell standard equality
eq {|Char|}      c          d         = c == d    -- Haskell standard equality
```

Generics defined for user-defined types Type-indexed functions (and also type-indexed types) can be defined over (possibly user-defined) datatypes, not only the structural representation types. This covers the cases for Int and Char, shown above. Additional cases such as the following are also possible:

```
eq {|Range|}   (R _ _)   (R _ _)   = True
```

for the user defined type

```
data Range = R Int Int
```

Constructor cases Datatypes which have a large number of constructors often 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 H∀SKELL 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

case ⟨*qcon*⟩

as illustrated in the following

freecollect {|**case** *Lambda*|} (*Lambda* (*v*, *t*) *e*) = *filter* (≠ *v*) (*freecollect* {|Expr|} *e*)

The case *freecollect* {|**case** *Lambda*|} will be applied only when the value of type Expr (from which *Lambda* is a constructor) encountered has the form *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.

3.3 Generic type signatures

The syntax of a generic type signature is as follows:

⟨*varid*⟩ {|⟨*gtyvars*⟩|} :: ⟨*gtypesig*⟩

Generic H∀SKELL supports dependency-style generic type signatures. A dependency-style type signature uses one or more *generic* and zero or more *non-generic* (or *parametric*) type variables, which are separated using a vertical bar, as type pattern. The pair of integers consisting of the number of generic and parametric type variables of a type-indexed function is called its *arity*. Generic type variables must always be of kind ∗, whereas parametric type variables may have different kinds.

The example signature of the equality function,

eq {|*a* :: ∗|} :: *a* → *a* → Bool

consists of one generic variable *a*, and no parametric variables. Other examples are the signatures of *map* and *collect*,

map   {|a_1 :: ∗, a_2 :: ∗|} :: (*map* {|a_1, a_2|}) ⇒ a_1 → a_2
collect {|*a* :: ∗ | *v* :: ∗|} :: (*collect* {|*a* | *v*|}) ⇒ *a* → [*v*]

with two generic and one generic plus one parametric variable, respectively. The precise meaning of generic and parametric type variables is discussed in EGH. Other than in EGH, the kind of the type variables can be omitted if it can be inferred.

The right-hand side of a dependency-style type signature is of the form

⟨*deps*⟩ ⇒ ⟨*type*⟩

a list of dependencies, followed by the *base type*.

Dependencies If a type-indexed function refers directly or indirectly to another type-indexed function in its definition, it is usually (exceptions: the type argument of the call is constant, or the other function is a generic abstraction) a *dependency* of the function.

The equality function, as given in the example, depends on itself, so the full type signature of *eq* is

$$eq \ \{\!| a :: * |\!\} :: (eq \ \{\!| a |\!\}) \Rightarrow a \rightarrow a \rightarrow \text{Bool}$$

Often, Generic H∀SKELL can infer dependencies. If no list of dependencies is given in a generic type signature, Generic H∀SKELL will try to infer the dependencies and complain if it does not succeed. Dependency inference is a rather experimental feature, and it is likely to be changed in future releases.

Note that in Generic H∀SKELL a dependency in a type signature must always include the variables, the EGH abbreviation of omitting the variables for functions of arity $\{\!| 1 \mid 0 |\!\}$ is not available.

In older releases, dependencies of generic functions had to be specified explicitly using the **dependency** keyword. This is still possible when generic functions are defined using kind-indexed types, but not needed when dependency-style type signatures are used.

Kind-indexed types In previous releases of Generic H∀SKELL, type signatures of generic functions made use of kind-indexed types. This style of generic programming is still supported. In fact, dependency-style type signatures are internally converted into applications of kind-indexed types. Kind-indexed types are thus no longer required, but still supported for backwards compatibility.

The function *eq* with the type signature given above behaves as if it had been defined using the kind-indexed type Eq:

$$\textbf{type } \text{Eq} \ \{\!| * |\!\} \qquad a = a \rightarrow a \rightarrow \text{Bool}$$
$$\textbf{type } \text{Eq} \ \{\!| k \rightarrow l |\!\} \ a =$$
$$\quad \textbf{forall } u \, . \, \text{Eq} \ \{\!| k |\!\} \ u \rightarrow \text{Eq} \ \{\!| l |\!\} \ (a \ u)$$
$$eq \ \{\!| t :: k |\!\} :: \text{Eq} \ \{\!| k |\!\} \ t$$

Similarly, the function *collect* could be defined using the following kind-indexed type Collect:

$$\textbf{type } \text{Collect} \ \{\!| * |\!\} \qquad a \ v = a \rightarrow [\, v \,]$$
$$\textbf{type } \text{Collect} \ \{\!| k \rightarrow l |\!\} \ a \ v =$$
$$\quad \textbf{forall } u \, . \, \text{Collect} \ \{\!| k |\!\} \ u \ v \rightarrow \text{Collect} \ \{\!| l |\!\} \ (a \ u) \ v$$
$$collect \ \{\!| t :: k |\!\} :: \textbf{forall } v \, . \, \text{Collect} \ \{\!| k |\!\} \ t \ v$$

Note that the type variable v that is parametric in the dependency-style type signature of *collect* is just passed on unchanged in the kind-indexed type, and is universally quantified at the outer level.

If a function depends on other functions except itself, a kind-indexed type – in contrast to a dependency-style type signature – is not sufficient. The dependencies have to be made explicit using the **dependency** keyword. For example, the hypothetical function

$$pretty \ \{\!| a :: * |\!\} :: (important \ \{\!| a |\!\}, pretty \ \{\!| a |\!\}) \Rightarrow a \rightarrow \text{String}$$

which pretty-prints the important parts of a data structure, can also be defined using the kind-indexed type

```
type Pretty {[∗]}      a = a → String
type Pretty {[k → l]} a =
    forall u . Important {[k]} u → Pretty {[k]} u → Pretty {[l]} (a u)
pretty {[t :: k]} :: Pretty {[k]} t
```

but then, the statement

```
dependency pretty ← important pretty
```

must be supplied in addition. In general, kind-indexed types are defined according to the syntax:

```
type ⟨Conid⟩ {[∗]}      t₁ ... tₙ = ⟨type⟩
type ⟨Conid⟩ {[k → l]} t₁ ... tₙ = ⟨type⟩
```

3.4 Generic application

A type-indexed function can be specialised to a function by applying it to a type. Generic application extends the syntax of expressions (⟨aexp⟩) as follows:

```
⟨aexp⟩ ::= ...
        |  ⟨varid⟩ {[⟨type⟩]}
```

The type argument must not contain universal quantifiers or class constraints. When a generic function is used, its type argument must always be supplied – it cannot (yet) be inferred.

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 (⟨gtycon⟩) is thus extended as follows:

```
⟨gtycon⟩ ::= ...
          |  ⟨Conid⟩ {[⟨kind⟩]}
```

Example   Given the datatype:

```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
```

The *map* function for BinTree is *map* {[BinTree]}. The type of *map* {[BinTree]} is $(a → b) →$ (BinTree a → BinTree b).

Short notation Type-indexed functions (that are not defined via generic abstraction, see Section 3.7) can be applied to type arguments of any kind. The dependencies for the type arguments are required in the same order as specified in the type signature or the **dependency** statement for the function. This mechanism is called *short notation* in EGH.

Other than in EGH, it is currently possible to use short notation with functions that have multiple dependencies. It is even possible to use short notation with functions that have inferred dependencies, although this is not recommended, as the order of dependencies then depends on the implementation of the inference algorithm, and this behaviour is likely to be disabled in future releases.

3.5 Local redefinition

It is possible to locally redefine the behaviour of a generic function, such as described in Chapter 8 of EGH. Local redefinition looks much like an ordinary **let** statement:

let $\langle varid \rangle$ $\{\!|\, a_0 \; a_1 \dots a_n \,|\!\} \dots = \dots$
$$\vdots$$
in $\dots$

The generic function referred to by $\langle varid \rangle$ must be in scope. Here, a_0, a_1, ..., a_n are all type variables. The variable a_0 is the variable on which the function is redefined, the arguments a_1, ..., a_n are local to the right-hand side, and only required if a_0 is of kind other than $*$.

Example Assume that *size* is a generic function of type

$$size \; \{\!|\, t :: * \,|\!\} :: (size \; \{\!|\, t \,|\!\}) \Rightarrow t \to \text{Int}$$

computing the size of a value. All base types and Unit return 0. In sums, the size of the particular alternative is chosen, and in products, the sizes of the components are added. This function is useless on types of kind $*$ without local redefinition, because it would always return 0. However,

let $size \; \{\!|\, a \,|\!\} = const \; 1$
in $(size \; \{\!|\, [[\text{Int}]] \,|\!\} \; [[1, 2, 3], [4, 5]],$
$\quad\;\, size \; \{\!|\, [[a]] \,|\!\} \quad [[1, 2, 3], [4, 5]],$
$\quad\;\, size \; \{\!|\, [a] \,|\!\} \quad\;\; [[1, 2, 3], [4, 5]],$
$\quad\;\, size \; \{\!|\, a \,|\!\} \quad\;\;\; [[1, 2, 3], [4, 5]])$

evaluates to $(0, 5, 2, 1)$.

3.6 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.

Note Between the Beryl and Coral releases, the syntax of default cases has been changed in a non backwards-compatible way. The original syntax [1] clashes with the more liberal syntax of generic abstractions and has therefore been removed. Instead, the syntax described in Chapter 14 of EGH is supported.

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

$$collect \; \{\!| a :: * \mid v :: * |\!\} :: (collect \; \{\!| a \mid v |\!\}) \Rightarrow a \rightarrow [\, v \,]$$

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

$$varcollect \; \{\!| a :: * |\!\} \qquad :: (varcollect \; \{\!| a \mid \mathrm{Var} |\!\}) \Rightarrow a \rightarrow [\, \mathrm{Var} \,]$$

by writing but a few lines:

$$varcollect \; \textbf{extends} \; collect$$
$$varcollect \; \{\!| \mathrm{Var} |\!\} \qquad v \qquad = [\, v \,]$$
$$varcollect \; \{\!| \mathrm{Prod} \; a \; b |\!\} \; (x\!:\!*\!:\!y) = varcollect \; \{\!| a |\!\} \; x \cup varcollect \; \{\!| b |\!\} \; y$$

The line containing the keyword **extends** is the default case, which has the effect of copying the code from *collect* into the new generic function *varcollect*. The line for *varcollect* $\{\!|\mathrm{Var}|\!\}$ specifies the desired additional functionality for type Var. The line for *varcollect* $\{\!|\mathrm{Prod} \; a \; b|\!\}$ overrides the functionality for Prod, using union instead of concatenation to accumulate the results.

3.7 Generic abstraction

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

$$\langle varid \rangle \; \{\!| \langle gtyvars \rangle |\!\} \; :: \; \langle gtypesig \rangle$$
$$\langle varid \rangle \; \{\!| t |\!\} \ldots \qquad = \langle exp \rangle$$

The difference with ordinary type-indexed functions is that the generic type arguments in the type signature may be of a kind other than $*$, and the type variable t must be of the same kind. Furthermore, a generic abstraction has exactly one case, of the given form.

Example An example is the so-called categorical strength:

$$strength \; \{\!| t :: * \rightarrow * |\!\} \; :: \; t \; a \rightarrow b \rightarrow t \; (a, b)$$
$$strength \; \{\!| t |\!\} \; ta \; b \qquad = map \; \{\!| t |\!\} \; (\lambda x \rightarrow (x, b)) \; ta$$

Deviating from EGH, the dependencies of generic abstractions are usually inferred when no explicit dependencies are provided in a type signature. The same holds for the kind of the type argument. It is, however, possible, to specify everything explicitly:

$$strength \; \{\!| t :: * \rightarrow * |\!\} :: (map \; \{\!| t, t |\!\}) \Rightarrow t \; a \rightarrow b \rightarrow t \; (a, b)$$

3.8 Type-indexed types

Warning   The implementation of type-indexed types has not been updated to support dependency-style definitions as used in EGH.

Type-indexed types [7] can be defined similar to type-indexed functions. A type-indexed type is defined by a generic kind signature, which defines its dependencies and kind.

$\langle deps \rangle \Rightarrow \langle kind \rangle$

The generic kind signature is followed by a collection of definitions. The right-hand side of such a definition consists of a constructor followed by a type:

$\langle Conid \rangle$ {|$\langle gtyvars \rangle$|} :: $\langle gkindsig \rangle$
type $\langle Conid \rangle$ {|$\langle stype \rangle$|} $t_1 \ldots t_n = \langle con \rangle \ \langle type \rangle$
type $\langle Conid \rangle$ {|$\langle stype \rangle$|} $t_1 \ldots t_n = \langle con \rangle \ \langle type \rangle$
$\vdots$

Only named type constructors are supported as $\langle stype \rangle$, and Con and Label do not get additional descriptor arguments on the type level. 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.

$\langle gtycon \rangle ::= \ldots$
$\quad\quad\quad | \ \langle Conid \rangle$ {|$\langle type \rangle$|}

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

FMap {|$a :: *$|} :: (FMap {|a|}) $\Rightarrow * \rightarrow *$
type FMap {|Unit|} $v = FMapUnit$ (Maybe v)
type FMap {|Sum|} $fma\ fmb\ v = FMapSum\ (fma\ v, fmb\ v)$
type FMap {|Prod|} $fma\ fmb\ v = FMapProd\ (fma\ (fmb\ v))$

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

type Lookup {|$*$|} $t = $ **forall** v . FMap {|t|} $v \rightarrow t \rightarrow$ Maybe v
type Lookup {|$k \rightarrow l$|} $t = $ **forall** a . Lookup {|k|} $a \rightarrow$ Lookup {|l|} $(t\ a)$

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

$lookup$ {|$t :: k$|} :: Lookup {|k|} t
$lookup$ {|Unit|} $(FMapUnit\ fm)$ $Unit$ $= fm$
$lookup$ {|Sum $a\ b$|} $(FMapSum\ (fma, fmb))\ (Inl\ a)$ $= lookup$ {|a|} $fma\ a$
$lookup$ {|Sum $a\ b$|} $(FMapSum\ (fma, fmb))\ (Inr\ b)$ $= lookup$ {|b|} $fmb\ b$
$lookup$ {|Prod $a\ b$|} $(FMapProd\ fma)$ $(a{:}{*}{:}b) = $ **do** $fmb \leftarrow lookup$ {|a|} $fma\ a$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad lookup$ {|b|} $fmb\ b$

3.9 Generic views

Type-indexed functions and type-indexed types become generic, because Generic H∀SKELL internally encodes data types as sums of products. Besides this standard structural representation, this release of Generic H∀SKELL also implements the list-like sums and products view and the fixed-point view. Several other generic views exist, such as the *Scrap Your Boilerplate* (SYB) view and the *balanced sums and products* view, but these are not included in this release.

Generic views for generic functions   With generic views for generic functions [4] it becomes possible to define generic functions using different structural representations. Such functions are sometimes harder or even impossible to define using the standard view of sums of products. Thus, generic views increase the expressive power of Generic H∀SKELL.

A generic function can use a different generic view than the standard view by specifying it using the keyword **viewed** in its generic type signature as follows:

⟨*varid*⟩ {|⟨*gtyvars*⟩ **viewed** ⟨*gview*⟩|} :: ⟨*gtypesig*⟩

The function *children* is an example of a generic function defined using the fixed-point view.

$children${|$a :: * $**viewed** Fix|} :: ($collect${|$a \mid c$|}) ⇒ $a → [a]$,
$children$ {|Fix f|} ($In\ r$) = **let** $collect${|a|} $x = [x]$ **in** $collect${|$f\ a$|} r

The structural representation type for the fixed-point view uses the type Fix to encode the recursive occurrences of data types.

data Fix $f = In\ (f\ (Fix\ f))$

For instance, the Tree data type can be "viewed" as Fix applied to the base functor of Tree.

data Tree a $= Leaf\ a \mid Branch\ (Tree\ a)\ (Tree\ a)$
data TreeBase $a\ r = LeafF\ a \mid BranchF\ r\ r$
type TreeF a $= Fix\ (TreeBase\ a)$

where the pattern functor TreeBase abstracts from the recursive occurrences of Tree a in the type argument r.

Instead of using a non-generic type parameter in the dependency on collect, we can alternatively define the generic type signature of *children* using a kind-indexed type:

dependency *children* ← *collect*
type Children{|∗|} $a = a → [a]$
type Children{|$k → l$|} $a =$
   forall x . (**forall** c . Collect{|k|} $x\ c$) → Children{|l|} ($a\ x$)
$children${|$a :: k $**viewed** Fix|} :: Children{|k|} a.

Note that in this example Collect is the kind-indexed type of *collect*.

Since the fixed-point view only uses a single type, i.e. the Fix type, generic functions defined using the fixed-point view only need to define a single arm. In the definition of *children* we use the function *collect* to collect the values of type Fix (TreeBase a) from the pattern functor TreeBase a (Fix (TreeBase a)), i.e. the recursive occurrences, obtaining the top-level recursive occurrences of the data type.

Generic views for generic types Besides using different structural representations in the definitions of type-indexed types, this release also supports generic views for generic types [12]. As with generic views for generic types, this increases the expressive power of Generic H∀SKELL.

Similar to a generic function, a generic type can use a different generic view than the standard view by specifying it using the keyword **viewed** in its generic kind signature:

⟨*Conid*⟩ {|⟨*gtyvars*⟩ **viewed** ⟨*gview*⟩|} :: ⟨*gkindsig*⟩

For example, the type Loc is a generic type that is defined using the fixed-point view. This generic type is used in the implementation of the generic zipper, which can be found in the examples directory of this release. The generic zipper is a data structure for tree traversals in which we can only traverse to to subtrees, i.e. recursive occurrences. Therefore, use of the fixed-point viewis required in its definition:

Loc {|a :: ∗ **viewed** Fix|}   :: (GID, Path) ⇒ ∗
type Loc {|Fix|} ($gidF$ :: ∗ → ∗) ($pathF$ :: ∗ → ∗ → ∗) = . . .

It is advisable to provide the dependency argument of a type-indexed type definition with the appropriate kinds to guide the Generic H∀SKELL compiler in kind inferencing.

The generic type Loc is used in the type of the navigation functions which allow us to navigate through a tree. For example, the generic function *down* moves the point of focus to the leftmost child of the current point of focus. Again, this generic function is defined using the fixed-point view:

down {|a :: ∗ **viewed** Fix|} :: Loc {|a|} → Loc {|a|}
down {|Fix f|} x = . . .

The advantage of using the fixed-point view on the type level and the value level is that the user is no longer required to provide the pattern functor explicitly when using the generic zipper.

3.10 Specialisation

Generic functions and generic types 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 or generic type is used with, and then generates the set of specialised versions for that function or type in the output file.

Specialisations are always generated locally per module. Thus, a generic function or generic type 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 fixed-point is reached. If the compiler is working correctly, the fixed-point calculation always terminates.

3.11 Generated function naming

The Generic H∀SKELL programmer must be aware that the generated Haskell code is polluted with additional names corresponding to instances of generic functions. These may clash with a programmer's own function names. Fortunately, this is highly unlikely as the generated names are rather complicated, encoding details such as module and type names. Names generated by the compiler all begin with `gh?_` or `GH?_`, where `?` is an arbitrary letter.

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.

3.12 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 functions, 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 function, or to limit the constructors visible for a type-indexed type.

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

At the moment, it is not possible to define a type-indexed function or type-indexed type across modules. However, one 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.

3.13 Haskell compatibility

Generic H∀SKELL parses all Haskell programs, except in the following instances:
- The tokens **forall**, **extends**, and **dependency** are additional keywords in Generic H∀SKELL.
- 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

$$\textbf{do}\ \{\ \sqcup\ [x] \leftarrow \mathit{action}; \mathit{return}\ x\ \}$$

4 Library

4.1 Introduction

The Generic H∀SKELL system comes with a library of useful generic functions. These are summarised below; for the details, consult the library itself (in subdirectory `lib`). We give the dependency-style type signatures of the generic functions, as well as the instantiated types for kind $*$ and kind $* \rightarrow *$, 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 `GH.Library.Bounds`

$$
\begin{array}{ll}
gminBound \ \{\!\|t :: *\|\!\} & :: (gminBound \ \{\!\|t\|\!\}) \Rightarrow t \\
gmaxBound \ \{\!\|t :: *\|\!\} & :: (gmaxBound \ \{\!\|t\|\!\}) \Rightarrow t \\
gminBound, gmaxBound \ \{\!\|t :: *\|\!\} & :: t \\
gminBound, gmaxBound \ \{\!\|t :: * \rightarrow *\|\!\} & :: a \rightarrow t \ a
\end{array}
$$

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

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

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 [11, Appendix D].

4.3 Module `GH.Library.Collect`

The functions in this module collect information about types, their constructors and their labels.

$$
\begin{array}{ll}
constructorOf \ \{\!\|t :: *\|\!\} & :: (constructorOf \ \{\!\|t\|\!\}) \Rightarrow t \rightarrow \text{ConDescr} \\
constructorOf \ \{\!\|t :: *\|\!\} & :: t \rightarrow \text{ConDescr} \\
constructorOf \ \{\!\|t :: * \rightarrow *\|\!\} & :: (a \rightarrow \text{ConDescr}) \rightarrow t \ a \rightarrow \text{ConDescr}
\end{array}
$$

The function *constructorOf* returns a description of the topmost constructor in a value.

$$
\begin{array}{ll}
constructors \ \{\!\|t :: *\|\!\} & :: (constructors \ \{\!\|t\|\!\}) \Rightarrow [\text{ConDescr}] \\
constructors \ \{\!\|t :: *\|\!\} & :: [\text{ConDescr}] \\
constructors \ \{\!\|t :: * \rightarrow *\|\!\} & :: [\text{ConDescr}] \rightarrow [\text{ConDescr}]
\end{array}
$$

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

$$labelsOf \ \{|t :: *|\} \qquad :: (labelsOf \ \{|t|\}) \Rightarrow t \rightarrow [\text{LabelDescr}]$$
$$labelsOf \ \{|t :: *|\} \qquad :: t \rightarrow [\text{LabelDescr}]$$
$$labelsOf \ \{|t :: * \rightarrow *|\} \ :: (a \rightarrow \text{LabelDescr}) \rightarrow t \ a \rightarrow \text{LabelDescr}$$

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

$$labels \ \{|t :: *|\} \qquad :: (labels \ \{|t|\}) \Rightarrow [\text{LabelDescr}]$$
$$labels \ \{|t :: *|\} \qquad :: [\text{LabelDescr}]$$
$$labels \ \{|t :: * \rightarrow *|\} \ :: [\text{LabelDescr}] \rightarrow [\text{LabelDescr}]$$

The function *labels* returns a list of descriptions of labels for a type, or the empty list when the datatype has no constructors with labels.

$$constructorsAndLabels \ \{|t :: *|\} \qquad :: (constructorsAndLabels \ \{|t|\}, labels \ \{|t|\}) \Rightarrow$$
$$[(\text{ConDescr}, [\text{LabelDescr}])]$$
$$constructorsAndLabels \ \{|t :: *|\} \qquad :: [(\text{ConDescr}, [\text{LabelDescr}])]$$
$$constructorsAndLabels \ \{|t :: * \rightarrow *|\} :: [(\text{ConDescr}, [\text{LabelDescr}])] \rightarrow$$
$$[\text{LabelDescr}] \qquad \rightarrow$$
$$[(\text{ConDescr}, [\text{LabelDescr}])]$$

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

The definitions of ConDescr and LabelDescr are given in Section 3.2.

4.4 Module `GH.Library.Compare`

$$gcompare \ \{|t :: *|\} \qquad :: (gcompare \ \{|t|\}) \Rightarrow t \rightarrow t \rightarrow \text{Ordering}$$
$$gcompare \ \{|t :: *|\} \qquad :: t \rightarrow t \rightarrow \text{Ordering}$$
$$gcompare \ \{|t :: * \rightarrow *|\} \ :: (a \rightarrow a \rightarrow \text{Ordering}) \rightarrow t \ a \rightarrow t \ a \rightarrow \text{Ordering}$$

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

4.5 Module `GH.Library.DeepSeq`

$$dSeq \ \{|t :: * \mid b :: *|\} :: (dSeq \ \{|t \mid b|\}) \Rightarrow t \rightarrow b \rightarrow b$$

The function *dSeq* is a variant of the standard function *seq*, which evaluates its first argument completely before returning the second.

This module also defines the type class DeepSeq

class DeepSeq a **where**
  $deepSeq :: a \rightarrow b \rightarrow b$

and the operator

$$(\$!!) :: (\mathsf{DeepSeq}\ a) \Rightarrow (a \rightarrow b) \rightarrow a \rightarrow b$$

for completely strict application, a variant of ($!).

Class instances for $\mathsf{DeepSeq}$ are provided, using the generic function *dSeq*, for a few standard datatypes.

4.6 Module `GH.Library.Enum`

$$
\begin{aligned}
&empty\ \{\!|\ t :: * \ |\!\} && :: (empty\ \{\!|\ t\ |\!\}) \Rightarrow t \\
&empty\ \{\!|\ t :: * \ |\!\} && :: t \\
&empty\ \{\!|\ t :: * \rightarrow * \ |\!\} :: a \rightarrow t\ a
\end{aligned}
$$

The *empty* function generates a default value of a datatype. For datatypes with multiple constructors, the leftmost constructor is preferred.

$$
\begin{aligned}
&enum\ \{\!|\ t :: * \ |\!\} && :: (enum\ \{\!|\ t\ |\!\}) \Rightarrow [\,t\,] \\
&enum\ \{\!|\ t :: * \ |\!\} && :: [\,t\,] \\
&enum\ \{\!|\ t :: * \rightarrow * \ |\!\} :: [\,a\,] \rightarrow [\,t\ a\,]
\end{aligned}
$$

The function *enum* enumerates all values of a datatype in a (possibly infinite) list. For infinite datatypes, the function applies diagonalisation such that each value appears at a finite position of the list.

4.7 Module `GH.Library.Eq`

$$
\begin{aligned}
&eq\ \{\!|\ t :: * \ |\!\} && :: (enum\ \{\!|\ t\ |\!\},\ eq\ \{\!|\ t\ |\!\}) \Rightarrow t \rightarrow t \rightarrow \mathsf{Bool} \\
&eq\ \{\!|\ t :: * \ |\!\} && :: t \rightarrow t \rightarrow \mathsf{Bool} \\
&eq\ \{\!|\ t :: * \rightarrow * \ |\!\} :: [\,a\,] \rightarrow (a \rightarrow a \rightarrow \mathsf{Bool}) \rightarrow t\ a \rightarrow t\ a \rightarrow \mathsf{Bool}
\end{aligned}
$$

The function *eq* is the generic version of (==) in the Eq class. It depends on *enum* so that functions with finite domain can be compared.

$$
\begin{aligned}
&neq\ \{\!|\ t :: * \ |\!\} && :: (enum\ \{\!|\ t\ |\!\},\ eq\ \{\!|\ t\ |\!\}) \Rightarrow t \rightarrow t \rightarrow \mathsf{Bool} \\
&neq\ \{\!|\ t :: * \ |\!\} && :: t \rightarrow t \rightarrow \mathsf{Bool} \\
&neq\ \{\!|\ t :: * \rightarrow * \ |\!\} :: [\,a\,] \rightarrow (a \rightarrow a \rightarrow \mathsf{Bool}) \rightarrow t\ a \rightarrow t\ a \rightarrow \mathsf{Bool}
\end{aligned}
$$

The function *neq* is the generic version of ($\neq$). It is defined to be the negation of *eq*.

4.8 Module `GH.Library.Map`

$$
\begin{aligned}
&gmap\ \{\!|\ t_1 :: *,\ t_2 :: * \ |\!\} :: (gmap\ \{\!|\ t_1, t_2\ |\!\}) \Rightarrow t_1 \rightarrow t_2 \\
&gmap\ \{\!|\ t :: * \ |\!\} && :: t \rightarrow t \\
&gmap\ \{\!|\ t :: * \rightarrow * \ |\!\} && :: (a \rightarrow b) \rightarrow t\ a \rightarrow t\ b
\end{aligned}
$$

The function *gmap* is the generic version of *fmap* in the $\mathsf{Functor}$ class.

4.9 Module `GH.Libary.MapM`

$mapMl \ \{t_1 :: *, t_2 :: * \mid m :: * \rightarrow *\} :: (mapMl \ \{t_1, t_2 \mid m\}, \mathsf{Functor}\ m, \mathsf{Monad}\ m) \Rightarrow t_1 \rightarrow m\ t_2$
$mapMr \ \{t_1 :: *, t_2 :: * \mid m :: * \rightarrow *\} :: (mapMr \ \{t_1, t_2 \mid m\}, \mathsf{Functor}\ m, \mathsf{Monad}\ m) \Rightarrow t_1 \rightarrow m\ t_2$
$mapMl, mapMr \ \{t :: *\}$ $:: (\mathsf{Functor}\ m, \mathsf{Monad}\ m) \Rightarrow t \rightarrow m\ t$
$mapMl, mapMr \ \{t :: * \rightarrow *\}$ $:: (\mathsf{Functor}\ m, \mathsf{Monad}\ m) \Rightarrow (a \rightarrow m\ b) \rightarrow t\ a \rightarrow m\ (t\ b)$

These are the generic versions of the monadic map $mapM$ in the Prelude. The function $mapMl$ 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.

4.10 Module `GH.Library.ReadShow`

$gshowsPrec \ \{t :: *\}$ $:: (gshowsPrec \ \{t\}) \Rightarrow \mathrm{Bool} \rightarrow \mathrm{Int} \rightarrow t \rightarrow \mathrm{ShowS}$
$greadsPrec \ \{t :: *\}$ $:: (greadsPrec \ \{t\}) \Rightarrow \mathrm{Bool} \rightarrow \mathrm{Int} \rightarrow \mathrm{ReadS}\ t$
$gshowsPrec \ \{t :: *\}$ $:: \mathrm{Bool} \rightarrow \mathrm{Int} \rightarrow t \rightarrow \mathrm{ShowS}$
$gshowsPrec \ \{t :: * \rightarrow *\}$ $:: (\mathrm{Bool} \rightarrow \mathrm{Int} \rightarrow a \rightarrow \mathrm{ShowS}) \rightarrow \mathrm{Bool} \rightarrow \mathrm{Int} \rightarrow t\ a \rightarrow \mathrm{ShowS}$
$greadsPrec \ \{t :: *\}$ $:: \mathrm{Bool} \rightarrow \mathrm{Int} \rightarrow \mathrm{ReadS}\ t$
$greadsPrec \ \{t :: * \rightarrow *\}$ $:: (\mathrm{Bool} \rightarrow \mathrm{Int} \rightarrow \mathrm{ReadS}\ a) \rightarrow \mathrm{Bool} \rightarrow \mathrm{Int} \rightarrow \mathrm{ReadS}\ (t\ a)$

These functions are generic versions of *show* and *read* (in classes Show and Read). The first 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*. The second argument of type Int specifies the precedence level.

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

$gshow \ \{t :: *\}$ $:: (gshowsPrec \ \{t\}) \Rightarrow t \rightarrow \mathrm{String}$
$gshow1 \ \{t :: * \rightarrow * \mid a :: *\} :: (gshowsPrec \ \{t\}, \mathsf{Show}\ a) \Rightarrow t\ a \rightarrow \mathrm{String}$
$gread \ \{t :: *\}$ $:: (greadsPrec \ \{t\}) \Rightarrow \mathrm{String} \rightarrow t$
$gread1 \ \{t :: * \mid a :: *\}$ $:: (greadsPrec \ \{t\}, \mathsf{Read}\ a) \Rightarrow \mathrm{String} \rightarrow t\ a$
$gshow \ \{t :: *\}$ $:: t \rightarrow \mathrm{String}$
$gshow1 \ \{t :: * \rightarrow *\}$ $:: \mathsf{Show}\ a \Rightarrow t\ a \rightarrow \mathrm{String}$
$gread \ \{t :: *\}$ $:: \mathrm{String} \rightarrow t$
$gread1 \ \{t :: * \rightarrow *\}$ $:: \mathsf{Read}\ a \Rightarrow \mathrm{String} \rightarrow t\ a$

4.11 Module `GH.Library.Reduce`

$rreduce \ \{t :: * \mid b :: *\} :: (rreduce \ \{t \mid b\}) \Rightarrow t \rightarrow b \rightarrow b$
$lreduce \ \{t :: * \mid b :: *\} :: (lreduce \ \{t \mid b\}) \Rightarrow b \rightarrow t \rightarrow b$
$rreduce \ \{t :: *\}$ $:: t \rightarrow b \rightarrow b$
$rreduce \ \{t :: * \rightarrow *\}$ $:: (a \rightarrow b \rightarrow b) \rightarrow t\ a \rightarrow b \rightarrow b$
$lreduce \ \{t :: *\}$ $:: b \rightarrow t \rightarrow b$
$lreduce \ \{t :: * \rightarrow *\}$ $:: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow t\ a \rightarrow b$

The function *rreduce* is a generic version of *foldr* (note the reversed order of the last two arguments!), while *lreduce* is a generic *foldl* ([2, Section 5.4] and [9, Section 12.2]).

$$crush \ \{ t :: * \to * \mid a :: * \} :: (lreduce \ \{ t \mid a \}) \Rightarrow (a \to a \to a) \to a \to t \ a \to a$$
$$crush \ \{ t :: * \to * \} \qquad :: (a \to a \to a) \to a \to t \ a \to a$$

The function *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:

$$gsum, gproduct \ \{ t :: * \to * \mid a :: * \} :: (lreduce \ \{ t \mid a \}, \mathsf{Num} \ a) \Rightarrow t \ a \to a$$
$$gand, gor \ \{ t :: * \} \qquad\qquad :: (lreduce \ \{ t \mid \mathsf{Bool} \}) \Rightarrow t \to \mathsf{Bool} \to \mathsf{Bool}$$
$$flatten \ \{ t :: * \to * \mid a :: * \} \qquad :: (rreduce \ \{ t \mid [a] \}) \Rightarrow t \ a \to [a]$$
$$count \ \{ t :: * \to * \mid a :: * \} \qquad :: (rreduce \ \{ t \mid \mathrm{Int} \}) \Rightarrow t \ a \to \mathrm{Int}$$
$$comp \ \{ t :: * \to * \mid a :: * \} \qquad :: (lreduce \ \{ t \mid a \to a \}) \Rightarrow t \ (a \to a) \to (a \to a)$$
$$gconcat \ \{ t :: * \to * \mid a :: * \} \qquad :: (lreduce \ \{ t \mid [a] \}) \Rightarrow t \ [a] \to [a]$$
$$gall \ \{ t :: * \to * \mid a :: * \} \qquad :: (lreduce \ \{ t \mid \mathsf{Bool} \}) \Rightarrow (a \to \mathsf{Bool}) \to t \ a \to \mathsf{Bool}$$
$$gany \ \{ t :: * \to * \mid a :: * \} \qquad :: (rreduce \ \{ t \mid \mathsf{Bool} \}) \Rightarrow (a \to \mathsf{Bool}) \to t \ a \to \mathsf{Bool}$$
$$gelem \ \{ t :: * \to * \mid a :: * \} \qquad :: (rreduce \ \{ t \mid \mathsf{Bool} \}, \mathsf{Eq} \ a) \Rightarrow a \to t \ a \to \mathsf{Bool}$$
$$gsum, gproduct \ \{ t :: * \to * \} \qquad :: \mathsf{Num} \ a \Rightarrow t \ a \to a$$
$$gand, gor \ \{ t :: * \to * \} \qquad :: t \ \mathsf{Bool} \to \mathsf{Bool}$$
$$flatten \ \{ t :: * \to * \} \qquad :: t \ a \to [a]$$
$$count \ \{ t :: * \to * \} \qquad :: t \ a \to \mathrm{Int}$$
$$comp \ \{ t :: * \to * \} \qquad :: t \ (a \to a) \to (a \to a)$$
$$gconcat \ \{ t :: * \to * \} \qquad :: t \ [a] \to [a]$$
$$gall, gany \ \{ t :: * \to * \} \qquad :: (a \to \mathsf{Bool}) \to t \ a \to \mathsf{Bool}$$
$$gelem \ \{ t :: * \to * \} \qquad :: \mathsf{Eq} \ a \Rightarrow a \to t \ a \to \mathsf{Bool}$$

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

4.12 Module GH.Library.Table

The module `Table` provides a type-indexed type and functions for building memo tables of functions.

This module has not been updated to dependency-style and EGH syntax, because type-indexed type support is currently lacking.

4.13 Module GH.Library.ZipWith

$$gzipWith \ \{ t_1 :: *, t_2 :: *, t_3 :: * \} :: (gzipWith \ \{ t_1, t_2, t_3 \}) \Rightarrow (t_1, t_2) \to \mathrm{Maybe} \ t_3$$
$$gzipWith \ \{ t :: * \} \qquad\qquad :: (t, t) \to \mathrm{Maybe} \ t$$
$$gzipWith \ \{ t :: * \to * \} \qquad :: ((a, b) \to \mathrm{Maybe} \ c) \to (t \ a, t \ b) \to \mathrm{Maybe} \ (t \ c)$$

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

$$\mathit{gunzipWith}\ \{\!|t_1 :: *, t_2 :: *, t_3 :: *|\!\} :: (\mathit{gunzipWith}\ \{\!|t_1, t_2, t_3|\!\}) \Rightarrow t_1 \rightarrow (t_2, t_3)$$
$$\mathit{gunzipWith}\ \{\!|t :: *|\!\} \qquad\qquad\quad :: t \rightarrow (t, t)$$
$$\mathit{gunzipWith}\ \{\!|t :: * \rightarrow *|\!\} \qquad\quad :: (a \rightarrow (b, c)) \rightarrow t\ a \rightarrow (t\ b, t\ c)$$

The function *gunzipWith* is a generic version of *unzip*.

$$\mathit{gzip}\quad \{\!|t :: * \rightarrow * \mid a :: *, b :: *|\!\} :: (\mathit{gzipWith}\quad \{\!|t, t, t|\!\}) \Rightarrow t\ a \rightarrow t\ b \rightarrow \mathrm{Maybe}\ (t\ (a, b))$$
$$\mathit{gunzip}\ \{\!|t :: * \rightarrow * \mid a :: *, b :: *|\!\} :: (\mathit{gunzipWith}\ \{\!|t, t, t|\!\}) \Rightarrow t\ (a, b) \rightarrow (t\ a, t\ b)$$

$$\mathit{gzip}\quad \{\!|t :: * \rightarrow *|\!\} \qquad\qquad\quad :: t\ a \rightarrow t\ b \rightarrow \mathrm{Maybe}\ (t\ (a, b))$$
$$\mathit{gunzip}\ \{\!|t :: * \rightarrow *|\!\} \qquad\qquad :: t\ (a, b) \rightarrow (t\ a, t\ b)$$

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

5 Future Work

The future of Generic H∀SKELL is uncertain. We have started designing a generic programming library in Haskell, which might replace Generic H∀SKELL to a certain extent. However, we might continue our work on Generic H∀SKELL, for example on:

- adding a type checker and better support for generic type inference
- dependency-style definitions of type-indexed types
- generic abstractions and local redefinitions on the type level
- ...

As we have not yet decided how the next major release of the Generic H∀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 have a look at <http://generic-haskell.org> or 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@cs.uu.nl`. To subscribe to the mailing list, point your browser to <https://mail.cs.uu.nl/mailman/listinfo/generic-haskell> and follow the instructions.

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. Generic application to type arguments of higher kind does not work for generic abstractions. For example, defining

   ```
   import GH.Library.Eq
   ```
   $myeq \ \{|a|\} :: a \rightarrow a \rightarrow \text{Bool}$
   $myeq \ \{|a|\} = eq \ \{|a|\}$

   you can call $eq \ \{|[\,]|\} \ eq_a$, but not $myeq \ \{|[\,]|\} \ eq_a$. However, you can use local redefinition and say

   let $eq \ \{|a|\} = eq_a$ **in** $myeq \ (\,[\,a\,]\,)$

3. The implementation of type-indexed types is lacking in several areas. For example, a type-indexed 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. Furthermore, dependency-style definitions of type-indexed types are not yet supported.

4. Hiding imports does not work properly at the moment. A solution is to use qualified identifiers to disambiguate name resolving.

5. . . .

6.4 Change log

Emerald (1.80) Generic views for generic types [12] are now supported. Furthermore, the implementation of type-indexed types is improved. This release also supports GHC 6.8.2 while still being backwards compatible with GHC 6.6.

Diamond (1.62) This version builds correctly under GHC 6.6. It also fixes bugs with *gread* and *gshow* when handling labeled records and infix constructors.

Diamond (1.60) Generic views as described in the generic views paper [4] are now supported. This release also corrects a few errors that occur with default cases.

Coral (1.42) Dependency-style definitions of type-indexed functions [10] are supported, i.e., generic functions can be written in a simpler and more natural style. In particular, type signatures of generic functions have become simpler – there is no need to define kind-indexed types any longer. Local redefinition, generic abstraction, and default cases are now implemented as described in "Exploring Generic Haskell" [9].

Beryl (1.23) Syntax for using Con and Label in generic functions has slightly changed. 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 Acknowledgements

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

6.6 Copyright information

`gh` – a compiler for Generic H∀SKELL.
Copyright © 2001 – 2005 The Generic H∀SKELL Team. Utrecht University
   This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WAR-RANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

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 J. Gibbons and J. Jeuring, editors, *Generic Programming*, IFIP, pages 21–47. Kluwer Academic Publishers, 2003.

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

[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] Stefan Holdermans, Johan Jeuring, Andres Löh and Alexey Rodriguez. Generic Views on data types. In Tarmo Uustalu, editor, *Mathematics of Program Construction*, volume 4014 of *LNCS*, pages 209–234. Springer-Verlag, 2006.

[5] Ralf Hinze and Johan Jeuring. Generic Haskell: applications. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 57–97. Springer-Verlag, 2003.

[6] Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory. In *Generic Programming, Advanced Lectures*, volume 2793 of *LNCS*, pages 1–56. Springer-Verlag, 2003.

[7] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *Proceedings of the 6th Mathematics of Program Construction Conference, MPC'02*, volume 2386 of *LNCS*, pages 148–174, 2002.

[8] Patrik Jansson and Johan 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.

[9] Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, September 2004.

[10] Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style Generic Haskell. In Olin Shivers, editor, *Proceedings of the International Conference on Functional Programming, ICFP'03*, pages 141–152. ACM Press, August 2003. Also appeared as Technical Report UU-CS-2003-022, Institute of Information and Computing Sciences, Utrecht University.

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

[12] Thomas van Noort. *Generic views for generic types*. Master's thesis, Utrecht University, 2008.