Generic HA∀SKELL 0.99
Amber Release

Andres Löh

November 7, 2001

Overview

- Stepwise introduction to the **Generic H∀SKELL** language
- A sample **Generic H∀SKELL** session
- A look at the compiler
  - What does it do?
  - problems and features
- Future work

Generic H∀SKELL– the language

- a superset of Haskell 98
- type-indexed values (generic functions)
- kind-indexed types
- type-indexed types (experimental feature)

MPC-style generic functions

- define functions on the structure of data types
- define functions for all types (of all kinds)
- MPC-style generic functions possess kind-indexed types

Structure of data types

$$
\begin{array}{lcl}
\textbf{data}\ A\ arg_1 \ldots arg_k & = & Con_1\ T_{1,1}\ T_{1,2} \ldots T_{1,n_1} \\
& | & Con_2\ T_{2,1}\ T_{2,2} \ldots T_{2,n2} \\
& \ldots & \\
& | & Con_m\ T_{m,1}\ T_{m,2} \ldots T_{m,n_m} \\
\textbf{data}\ A^\circ\ arg_1 \ldots arg_k & = & T_{1,1}\ \texttt{:*:}\ T_{1,2}\ \texttt{:*:} \ldots \texttt{:*:}\ T_{1,n_1} \\
& \texttt{:+:} & T_{2,1}\ \texttt{:*:}\ T_{2,2}\ \texttt{:*:} \ldots \texttt{:*:}\ T_{2,n2} \\
& \ldots & \\
& \texttt{:+:} & T_{m,1}\ \texttt{:*:}\ T_{m,2}\ \texttt{:*:} \ldots \texttt{:*:}\ T_{m,n_m}
\end{array}
$$

- Both `:+:` and `:*:` are right associative.
- A constructor without elements is replaced by $Unit$.

$$
\begin{array}{lcl}
\textbf{data}\ List\ a & = & Nil \mid Cons\ a\ (List\ a) \\
\textbf{data}\ List^\circ\ a & = & Unit\ \texttt{:+:}\ a\ \texttt{:*:}\ List\ a
\end{array}
$$

Structure of data types – continued

$$
\begin{array}{lcl}
\textbf{data}\ a\ \texttt{:+:}\ b & = & Inl\ a \mid Inr\ b \\
\textbf{data}\ a\ \texttt{:*:}\ b & = & a\ \texttt{:*:}\ b \\
\textbf{data}\ Unit & = & Unit
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{data}\ Con\ a & = & Con\ ConDescr\ a \\
\textbf{data}\ Label\ a & = & Label\ LabelDescr\ a
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{data}\ List\ a & = & Nil \mid Cons\{\, hd :: a, tl :: List\ a \,\} \\
\textbf{data}\ List^{\circ}\ a & = & Con\ \boxed{\texttt{Nil}}\ Unit \\
& \texttt{:+:} & Con\ \boxed{\texttt{Cons}}\ (Label\ \boxed{\texttt{hd}}\ a\ \texttt{:*:}\ Label\ \boxed{\texttt{tl}}\ (List\ a))
\end{array}
$$

Structure of data types – continued

$$
\begin{array}{lcll}
\textbf{data}\ \mathit{ConDescr} & = & \mathit{ConDescr} & \{\ \mathit{conName} :: \mathit{String} \\
 & & & ,\ \mathit{conType} :: \mathit{String} \\
 & & & ,\ \mathit{conArity} :: \mathit{Int} \\
 & & & ,\ \mathit{conLabels} :: \mathit{Bool} \\
 & & & ,\ \mathit{conFixity} :: \mathit{Fixity} \\
 & & & \} \\
\textbf{data}\ \mathit{LabelDescr} & = & \mathit{LabelDescr} & \{\ \mathit{labelName} :: \mathit{String} \\
 & & & ,\ \mathit{labelType} :: \mathit{String} \\
 & & & ,\ \mathit{labelStrict} :: \mathit{Bool} \\
 & & & \}
\end{array}
$$

Structure of data types – more examples

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Fork}\ a & = & \mathit{Fork}\ a\ a \\
\textbf{data}\ \mathit{Perfect}\ a & = & \mathit{ZeroP}\ a \mid \mathit{SuccP}\ (\mathit{Perfect}\ (\mathit{Fork}\ a)) \\
\textbf{data}\ \mathit{Fork}^{\circ}\ a & = & \mathit{Con}\ \boxed{\texttt{Fork}}\ (a\ \texttt{:*:}\ a) \\
\textbf{data}\ \mathit{Perfect}^{\circ}\ a & = & \mathit{Con}\ \boxed{\texttt{ZeroP}}\ a \\
 & \texttt{:+:} & \mathit{Con}\ \boxed{\texttt{SuccP}}\ (\mathit{Perfect}\ (\mathit{Fork}\ a))
\end{array}
$$

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Tree}\ a\ b & = & \mathit{Leaf}\ a \\
 & \mid & \mathit{Node}\ (\mathit{Tree}\ a\ b)\ b\ (\mathit{Tree}\ a\ b) \\
\textbf{data}\ \mathit{Tree}^{\circ}\ a\ b & = & \mathit{Con}\ \boxed{\texttt{Leaf}}\ a \\
 & \texttt{:+:} & \mathit{Con}\ \boxed{\texttt{Node}}\ (\mathit{Tree}\ a\ b\ \texttt{:*:}\ b\ \texttt{:*:}\ \mathit{Tree}\ a\ b)
\end{array}
$$

Generic functions in Generic H∀SKELL

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Bit} & = & \texttt{O} \mid \texttt{I} \\
\quad \text{-- } \mathit{encode}\{\!|\, t \,|\!\}\ ::\ t \longrightarrow [\mathit{Bit}] & & \\
\mathit{encode}\{\!|\, \mathit{Unit} \,|\!\} & = & \mathit{const}\ [\,] \\
\mathit{encode}\{\!|\, \texttt{:+:} \,|\!\}\ \mathit{eA}\ \mathit{eB}\ (\mathit{Inl}\ a) & = & \texttt{O} : \mathit{eA}\ a \\
\mathit{encode}\{\!|\, \texttt{:+:} \,|\!\}\ \mathit{eA}\ \mathit{eB}\ (\mathit{Inr}\ b) & = & \texttt{I} : \mathit{eB}\ b \\
\mathit{encode}\{\!|\, \texttt{:*:} \,|\!\}\ \mathit{eA}\ \mathit{eB}\ (a\ \texttt{:*:}\ b) & = & \mathit{eA}\ a +\!\!+ \mathit{eB}\ b \\
\mathit{encode}\{\!|\, \mathit{Con}\ c \,|\!\}\ \mathit{eB}\ (\mathit{Con}\ \_\ b) & = & \mathit{eB}\ b \\
\mathit{encode}\{\!|\, \mathit{Label}\ l \,|\!\}\ \mathit{eB}\ (\mathit{Label}\ \_\ b) & = & \mathit{eB}\ b \\
\mathit{encode}\{\!|\, \mathit{Int} \,|\!\} & = & \mathit{primEncodeInt} \\
\mathit{encode}\{\!|\, \mathit{Char} \,|\!\} & = & \mathit{primEncodeChar}
\end{array}
$$

Generic functions in Generic H∀SKELL– continued

$$
\begin{array}{lcl}
\text{-- } decode\{\!|\, t \,|\!\}\ ::\ [Bit] \rightarrow (t, [Bit]) & & \\
decode\{\!|\, Unit \,|\!\}\ bl & = & (Unit, bl) \\
decode\{\!|\, \texttt{:+:} \,|\!\}\ dA\ dB\ (\texttt{O} : bl) & = & \textbf{let}\ (a, bl') = dA\ bl\ \textbf{in}\ (Inl\ a, bl') \\
decode\{\!|\, \texttt{:+:} \,|\!\}\ dA\ dB\ (\texttt{I} : bl) & = & \textbf{let}\ (b, bl') = dB\ bl\ \textbf{in}\ (Inr\ b, bl') \\
decode\{\!|\, \texttt{:*:} \,|\!\}\ dA\ dB\ bl & = & \textbf{let}\ (a, bl') = dA\ bl \\
 & & \quad\ (b, bl'') = dB\ bl' \\
 & & \textbf{in}\ (a\ \texttt{:*:}\ b, bl'') \\
decode\{\!|\, Con\ c \,|\!\}\ dB\ bl & = & \textbf{let}\ (b, bl') = dB\ bl\ \textbf{in}\ (Con\ c\ b, bl') \\
decode\{\!|\, Label\ l \,|\!\}\ dB\ bl & = & \textbf{let}\ (b, bl') = dB\ bl\ \textbf{in}\ (Label\ l\ b, bl') \\
decode\{\!|\, Int \,|\!\} & = & primDecodeInt \\
decode\{\!|\, Char \,|\!\} & = & primDecodeChar
\end{array}
$$

The type of $encode$ and $decode$

The type of the generic functions $encode$ and $decode$ depends on the kind of their type argument.

$$\begin{array}{lcl} encode\{\!|\, t :: k \,|\!\} & :: & Encode\{\!|\, k \,|\!\}\ t \\ decode\{\!|\, t :: k \,|\!\} & :: & Decode\{\!|\, k \,|\!\}\ t \end{array}$$

The types $Unit$, Int, $Char$ have kind $\star$:

$$\begin{array}{lcl} Encode\{\!|\, \star \,|\!\}\ t & = & t \rightarrow [Bit] \\ Decode\{\!|\, \star \,|\!\}\ t & = & [Bit] \rightarrow (t, [Bit]) \end{array}$$

The types :+: and :*: have kind $\star \rightarrow \star \rightarrow \star$:

$$\begin{array}{lcl} Encode\{\!|\, \star \rightarrow \star \rightarrow \star \,|\!\}\ t & = & (a \rightarrow [Bit]) \rightarrow (b \rightarrow [Bit]) \\ & \rightarrow & (t\ a\ b \rightarrow [Bit]) \\ Decode\{\!|\, \star \rightarrow \star \rightarrow \star \,|\!\}\ t & = & ([Bit] \rightarrow (a, [Bit])) \rightarrow ([Bit] \rightarrow (b, [Bit])) \\ & \rightarrow & ([Bit] \rightarrow (t\ a\ b, [Bit])) \end{array}$$

Kind-indexed types in Generic H∀SKELL

$$
\begin{array}{lcl}
\textbf{type } \mathit{Encode}\{\!|\star|\!\}\ t & = & t \to [\mathit{Bit}] \\
\textbf{type } \mathit{Encode}\{\!|k \to l|\!\}\ t & = & \textbf{forall } a\ .\ \mathit{Encode}\{\!|k|\!\}\ t \to \mathit{Encode}\{\!|l|\!\}\ (t\ a) \\
\textbf{type } \mathit{Decode}\{\!|\star|\!\}\ t & = & ([\mathit{Bit}] \to (a, [\mathit{Bit}])) \\
\textbf{type } \mathit{Decode}\{\!|k \to l|\!\}\ t & = & \textbf{forall } a\ .\ \mathit{Decode}\{\!|k|\!\}\ t \to \mathit{Decode}\{\!|l|\!\}\ (t\ a)
\end{array}
$$

Another example: $gmap$

$$
\begin{array}{lcl}
\textbf{type}\ Map\{\!|\star|\!\}\ s\ t & = & s \to t \\
\textbf{type}\ Map\{\!|k \to l|\!\}\ s\ t & = & \textbf{forall}\ a\ b\ .\ Map\{\!|k|\!\}\ a\ b \to Map\{\!|l|\!\}\ (s\ a)\ (t\ b) \\
gmap\{\!|t :: k|\!\} & :: & Map\{\!|k|\!\}\ t\ t \\
\ldots &&
\end{array}
$$

Type of $gmap$:

$$
\begin{array}{lcl}
gmap\{\!|Int|\!\} & :: & t \to t \\
gmap\{\!|[\,]|\!\} & :: & Map\{\!|\star \to \star|\!\}\ [\,]\ [\,] \\
 & = & \textbf{forall}\ a\ b\ .\ Map\{\!|\star|\!\}\ a\ b \to Map\{\!|\star|\!\}\ [a]\ [b] \\
 & = & \textbf{forall}\ a\ b\ .\ (a \to b) \to [a] \to [b] \\
gmap\{\!|Tree|\!\} & :: & (a \to c) \to (b \to d) \to Tree\ a\ b \to Tree\ c\ d \\
gmap\{\!|(,)|\!\} & :: & (a \to c) \to (b \to d) \to (a, b) \to (c, d)
\end{array}
$$

Another example: $gzipWith$

$$
\begin{array}{lcl}
\textbf{type}\ ZipWith\{\!|\star|\!\}\ t_1\ t_2\ t_3 & = & (t_1, t_2) \to Maybe\ t_3 \\
\textbf{type}\ ZipWith\{\!|k \to l|\!\}\ t_1\ t_2\ t_3 & = & \textbf{forall}\ a_1\ a_2\ a_3\ .\ ZipWith\{\!|k|\!\}\ a_1\ a_2\ a_3 \\
 & \to & ZipWith\{\!|l|\!\}\ (t_1\ a_1)\ (t_2\ a_2)\ (t_3\ a_3) \\
gzipWith\{\!|t :: k|\!\} & :: & ZipWith\{\!|k|\!\}\ t\ t\ t \\
gzipWith\{\!|Unit|\!\}\ \_ & = & return\ Unit \\
gzipWith\{\!|:+:|\!\}\ zA\ zB\ (Inl\ a_1, Inl\ a_2) & = & zA\ (a_1, a_2) \gg\!= return\ .\ Inl \\
gzipWith\{\!|:+:|\!\}\ zA\ zB\ (Inr\ b_1, Inr\ b_2) & = & zB\ (b_1, b_2) \gg\!= return\ .\ Inr \\
gzipWith\{\!|:+:|\!\}\ \_\ \_\ \_ & = & Nothing \\
gzipWith\{\!|:*:|\!\}\ zA\ zB\ (a_1 :*: a_2, b_1 :+: b_2) & = & \textbf{do}\ a \leftarrow zA\ (a_1, a_2) \\
 & & \quad b \leftarrow zB\ (b_1, b_2) \\
 & & \quad return\ (a :*: b) \\
gzipWith\{\!|Con\ c|\!\}\ zB\ (Con\ \_\ b_1, Con\ \_\ b_2) & = & zB\ (b_1, b_2) \gg\!= return\ .\ Con\ c \\
gzipWith\{\!|Label\ l|\!\}\ zB\ (Label\ \_\ b_1, Label\ \_\ b_2) & = & zB\ (b_1, b_2) \gg\!= return\ .\ Label\ l
\end{array}
$$

Generic abstraction in Generic H∀SKELL

Generic functions can be defined (for types of a fixed kind) in terms of other generic functions.

$$
\begin{aligned}
&gzip\{\!|\, t :: \star \longrightarrow \star \,|\!\} :: t\ a \quad \longrightarrow \quad t\ b \longrightarrow Maybe\ (t\ (a, b)) \\
&gzip\{\!|\, t \,|\!\}\ as\ bs = gzipWith\{\!|\, t \,|\!\}\ return\ (as, bs)
\end{aligned}
$$

Type-indexed data types in Generic H∀SKELL

$$
\begin{array}{lcl}
\textbf{type}\ Table\{\!|\,Unit\,|\!\}\ v & = & TUnit\ v \\
\textbf{type}\ Table\{\!|\,:+:\,|\!\}\ ta\ tb\ v & = & TSum\ (ta\ v, tb\ v) \\
\textbf{type}\ Table\{\!|\,:*:\,|\!\}\ ta\ tb\ v & = & TProd\ (ta\ (tb\ v)) \\
\textbf{type}\ Table\{\!|\,Con\,|\!\}\ tb\ v & = & TCon\ (tb\ v) \\
\textbf{type}\ Table\{\!|\,Label\,|\!\}\ tb\ v & = & TLabel\ (tb\ v) \\
\ldots & & \\
\textbf{type}\ Apply\{\![\star]\!\}\ key & = & \textbf{forall}\ val\ .\ Table\{\!|\,key\,|\!\}\ val \rightarrow key \rightarrow val \\
\textbf{type}\ Apply\{\![k \rightarrow l]\!\}\ key & = & \textbf{forall}\ a\ .\ Apply\{\![k]\!\}\ a \rightarrow Apply\{\![l]\!\}\ (key\ a) \\
apply\{\!|\,key :: k\,|\!\} & :: & Apply\{\![k]\!\}\ key \\
apply\{\!|\,Unit\,|\!\}\ (TUnit\ t)\ \_ & = & t \\
apply\{\!|\,:+:\,|\!\}\ aA\ aB\ (TSum\ (t_1, t_2))\ (Inl\ k_1) & = & aA\ t_1\ k_1 \\
apply\{\!|\,:+:\,|\!\}\ aA\ aB\ (TSum\ (t_1, t_2))\ (Inr\ k_2) & = & aB\ t_2\ k_2 \\
apply\{\!|\,:*:\,|\!\}\ aA\ aB\ (TProd\ t)\ (k_1\ :*:\ k_2) & = & aB\ (aA\ t\ k_1)\ k_2 \\
\ldots & &
\end{array}
$$

Generic H∀SKELL summary

New constructions:

- Kind-indexed types
- Type-indexed valued
  - MPC-style generic definitions
  - Generic abstraction
- Type-indexed types
- Possibility to refer to generic entities using $\{\!|\cdot|\!\}$ and $\{\![\cdot]\!\}$.

A sample Generic H∀SKELL session – input file

```
module TestGH where

import ReadShow

data Bit                        =  O | I
data Fork a                     =  Fork a a
data Perfect a                  =  ZeroP a | SuccP (Perfect (Fork a))

type Map {[ * ]} t1 t2 = t1 -> t2
type Map {[ k -> l ]} t1 t2 = forall u1 u2. Map {[ k ]} u1 u2
                                         -> Map {[ l ]} (t1 u1) (t2 u2)

gmap {| t :: k |} :: Map {[ k ]} t t
gmap {| Unit |}                           = id
gmap {| :+: |}     gmapA gmapB (Inl a)    = Inl (gmapA a)
gmap {| :+: |}     gmapA gmapB (Inr b)    = Inr (gmapB b)
gmap {| :*: |}     gmapA gmapB (a :*: b)  = (gmapA a) :*: (gmapB b)
gmap {| Con c |}   gmapA       (Con d a)  = Con d (gmapA a)
gmap {| Label l |} gmapA       (Label _ a) = Label l (gmapA a)
```

A sample Generic H∀SKELL session – input file, continued

```
aperfecttree = SuccP (SuccP (ZeroP (Fork (Fork O I) (Fork O I))))
mappedtree   = gmap{| Perfect |} invert aperfecttree
  where invert O = I
        invert I = O

test1 = showsPrec'{| Perfect Bit |} False 0 aperfecttree ""
test2 = showsPrec'{| Perfect Bit |} False 0 mappedtree ""
```

A sample Generic H∀SKELL session – compiler call

```
> $GH_HOME/bin/gh -v -L $GH_HOME/lib TestGH
File TestGH.ghs read.
Looking for interface file GHPrelude.ghi ... read.
Looking for interface file ReadShow.ghi ... read.
Interface file TestGH.ghi written.
Haskell file TestGH.hs written.
> hugs -P"{Hugs}/lib:$GH_HOME/lib" TestGH.hs
__   __ __  __  ____   ___      ________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-1999
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Report bugs to: hugs-bugs@haskell.org
||   || Version: February 2000  ________________________________________

[...]

TestGH>
```

A sample Generic H∀SKELL session – testing the result file

```
TestGH> test1
"SuccP (SuccP (ZeroP (Fork (Fork O I) (Fork O I))))"
TestGH> test2
"SuccP (SuccP (ZeroP (Fork (Fork I O) (Fork I O))))"
TestGH>
```

The compiler . . .

. . . works as a preprocessor from **Generic H∀SKELL** to Haskell. It does:

- import generic functions from other modules by reading interface files
- infer kinds for user-defined data types
- generate structure types for user-defined data types
- generate mappings between a type and its structure type
- specialise generic entities to different types:
  - a specialisation for the structure type is generated
  - a wrapper transforms that into a function that works on the "real" type.
- calls to generic entities are replaced by calls to specialisations.
- an interface file is generated to support separate compilation of modules

It does *not*:

- interpret Haskell code, i.e. no type-checking etc.

An example specialisation

Let's have a look at a (slightly formatted) output file:

```
gmapPerfect__       :: (u1 -> u2) -> Perfect__ u1 -> Perfect__ u2
gmapPerfect__ gh_a = gmapSum (gmapCon (ConDescr "ZeroP" "Perfect" 1 False Nonfix
)
                                      (gh_a))
                             (gmapCon (ConDescr "SuccP" "Perfect" 1 False Nonfix)
                                      (gmapPerfect (gmapFork (gh_a))))

gmapPerfect         :: (u1 -> u2) -> Perfect u1 -> Perfect u2
gmapPerfect gh_a    = to (bimapFun (epPerfect) (epPerfect)) (gmapPerfect__ gh_a)

gmapSum                       :: (u1 -> u2) -> (a -> b) -> Sum u1 a -> Sum u2 b
gmapSum gmapA gmapB (Inl a) = Inl (gmapA a)
gmapSum gmapA gmapB (Inr b) = Inr (gmapB b)

gmapCon                       :: ConDescr -> (u1 -> u2) -> Con u1 -> Con u2
gmapCon c gmapA (Con d a)    = Con d (gmapA a)
```

What gets specialised?

	imported functions	local functions
imported types	no	yes
local types	yes	yes

Therefore,

module *MyModule* **where**
import *MyBrilliantGenericFunction*
import *MyVerySpecialDatatype*

does *not* work.

Library

Generic H∀SKELL ships (thanks to Jan de Wit) with a library of generic functions, such as

- functions to determine the lower and upper bound of a type (`Bounds`)
- functions to compare values (`Compare`, `Eq`)
- mapping functions (`Map`, `MapM`)
- functions to read and show values (`ReadShow`)
- functions to collect contructor and label information (`Collect`)
- . . .

The library will be extended in the future.

Future work

- module support
  - qualified names
  - import/export lists
  - better specialisation mechanism
- add type checking
- generic functions
  - local generic definitions
  - POPL-style generic definitions
  - mutually recursive generic definitions
- extended support for type-indexed data types
- fixpoints and views
- support for generic traversal functions

The final slide

The compiler (source distribution as well as binary distributions for Linux, Solaris, MacOS X, and Windows) and additional information is available from:

- The **Generic H∀SKELL** project page:
  `http://www.generic-haskell.org`

Thanks to the **Generic H∀SKELL** team: Dæv Clarke, Ralf Hinze, Johan Jeuring, Jan de Wit

Thanks to Simon Marlow and Sven Panne for the Haskell parser in `hslibs`.

Thanks to Ralf Hinze for the `frown :-(` parser generator.

Thanks to Arthur Baars and Doaitse Swierstra for the attribute grammar system.