Generic H∀SKELL
—bringing generic functions into the Haskell world

Software Generation Course

Andres Löh

May 28, 2001

Overview

This talk is in large parts based on the work of Johan Jeuring (Utrecht), Ralf Hinze (Utrecht, Bonn), and Patrik Jansson (Chalmers).

- PART I: What are generic functions?
  - The Haskell **deriving** construct . . .
  - . . . and its limitations
  - Polymorphism and Polytypism
  - Formalising generic functions.
- PART II: Implementing **Generic H∀SKELL**
  - How can a generic function be implemented?
  - What does the **Generic H∀SKELL** compiler do . . .
  - . . . and how?

PART I

Extending a type system

Haskell possesses a very strong static type system. Type systems help the programmer to write *correct* programs.

Still, the Haskell type system lacks some flexibility we would like to have.

This talk is about extending the language with the desired possibilities without losing the benefits of the type system.

Reminder: Haskell type classes

Haskell allows to overload functions (like equality, numeric operators, . . .) by making them members of type classes. A type class is given by its signature, a number of member functions together with their types.

This example is copied from the Haskell prelude.

$$
\begin{array}{l}
\textbf{class}\ Eq\ a\ \textbf{where} \\
\quad \begin{array}{lcl}
(\equiv) & :: & a \to a \to Bool \\
(\not\equiv) & :: & a \to a \to Bool \\
\multicolumn{3}{l}{\text{— Minimal complete definition: } (\equiv) \text{ or } (\not\equiv)} \\
x \equiv y & = & \neg\ (x \not\equiv y) \\
x \not\equiv y & = & \neg\ (x \equiv y)
\end{array}
\end{array}
$$

instance

If one wants to provide equality for a new datatype such as

$$\begin{array}{lcl}\textbf{data}\ \mathit{Tree}\ a & = & \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\ & \mid & \mathit{Leaf}\end{array}$$

then it can be made an instance of the Eq class:

$$\begin{array}{lcl}\multicolumn{3}{l}{\textbf{instance}\ \mathit{Eq}\ a \Rightarrow \mathit{Eq}\ (\mathit{Tree}\ a)\quad \textbf{where}} \\ \quad \mathit{Node}\ a\ x\ b \equiv \mathit{Node}\ c\ y\ d & = & a \equiv c \wedge x \equiv y \wedge b \equiv d \\ \quad \mathit{Leaf} \equiv \mathit{Leaf} & = & \mathit{True} \\ \quad \_ \equiv \_ & = & \mathit{False}\end{array}$$

Some remarks are in order:

- We need equality on a to define equality on $\mathit{Tree}\ a$.
- The definition is otherwise straight-forward.
- The class mechanism does not enforce a sensible definition (ad-hoc overloading).

deriving

Writing equality instances for many datatypes can be tedious. That is why Haskell provides a special construct **deriving**. The code

$$
\begin{array}{rcl}
\mathbf{data}\ \mathit{Tree}\ a & = & \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\
 & | & \mathit{Leaf} \\
 & & \quad \mathbf{deriving}\ \mathit{Eq}
\end{array}
$$

automatically generates the instance code. But how?

Obviously, there is some kind of intuition how to define equality on a datatype, but can this intuition be transformed into a rule?

Second example: The *Show* class

The function $show$ can be used to get a string representation of a type that is instance of class $Show$.

$$
\begin{array}{l}
\textbf{class } Show\ a\ \textbf{where} \\
\quad show \quad :: \quad a \to String \\
\quad \ldots
\end{array}
$$

Using **deriving**, one can again save the work to give instance declarations by hand. A representation like the one that is used inside Haskell programs is derived automatically. Again, there must be a rule that directs this generation process.

Informal descriptions of $\equiv$ and $show$

- $\equiv$

  We get two values of the same datatype.

  Datatypes have a set of constructor names.

  Compare if the two arguments belong to the same constructor. If they don't, return $False$. If they do, then look at the components that belong to that specific constructor.

  The respective components of the two arguments must be equal.

  We have to know how to compute equality for the types of the components. If one of these types is an argument to the original type (like the a in $Tree\ a$), we have to ensure that a belongs to class Eq as well.

- $show$

  We get a value of a datatype.

  Datatypes have a set of constructor names.

  The string representation begins with the name of the constructor.

  Then the string representations (results of the $show$ function) of the components of that constructor follow, separated by space.

  We have to know how to compute $show$ for the types of the components. If one of these types is an argument to the original type, we have to ensure that a belongs to class $Show$ as well.

Observations

There is a common structure to the two informal descriptions.

- We start with the constructors.
- We continue with the fields/components of the constructors.
- We recursively apply the function to be defined to the components.
- We thus require the function to be defined for the components as well.

Third example: The $\mathit{Functor}$ class

This is again copied from the Haskell prelude:

$$
\begin{array}{lcl}
\mathbf{class}\ \mathit{Functor}\ f\ \mathbf{where} & & \\
\quad \mathit{fmap} & :: & (a \to b) \to (f\ a \to f\ b) \\
\mathbf{instance}\ \mathit{Functor}\ [\,]\ \mathbf{where} & & \\
\quad \mathit{fmap} & = & \mathit{map} \\
\mathbf{instance}\ \mathit{Functor}\ \mathit{Maybe}\ \mathbf{where} & & \\
\quad \mathit{fmap}\ f\ \mathit{Nothing} & = & \mathit{Nothing} \\
\quad \mathit{fmap}\ f\ (\mathit{Just}\ x) & = & \mathit{Just}\ (f\ x)
\end{array}
$$

The class $\mathit{Functor}$ is different from the examples so far in that it is a *constructor class*. The instances are not types, but *type constructors* ("functions from types to types").

Deriving $\mathit{Functor}$?

We can also define fmap for the Tree type constructor.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ \mathit{Tree}\ \textbf{where} \\
\quad \mathit{fmap}\ f\ (\mathit{Node}\ a\ x\ b) \quad = \quad \mathit{Node}\ (\mathit{fmap}\ f\ a)\ (f\ x)\ (\mathit{fmap}\ f\ b)
\end{array}
$$

The idea of fmap seems to be to leave the "structure" of the value unchanged, but to apply the given function to the "elements" (of the list, of the tree, of the Maybe).

But Haskell does not allow to use **deriving** *for class* $\mathit{Functor}$*!*

In fact, Haskell defines the **deriving** mechanism only for a small, predefined set of classes (Eq,Ord,Enum,$\mathit{Bounded}$,Read,Show). But obviously, there are more functions that can be defined "intuitively" for many types or type constructors.

Polymorphism

Before we start formalising our intuitions and informal descriptions, let's have a look at polymorphism. A function like

$$\begin{array}{lcl} id & :: & a \to a \\ id\ x & = & x \end{array}$$

is called *polymorphic*. It can be used for any type a, because it does not assume anything about the type. So the *same* code can be used for many types.

Compare with type classes: There also a function can be used for many types, but they do *not* share the same code. In fact, the code may be completely unrelated!

Polytypism

The phenomenon we are discussing is in between these two extrema: Functions generated by the **deriving** construct, but also others like $fmap$ do not share the *same* code for different types, but are also not completely unrelated.

They are dependent on the structure of types! Such functions are called

- polytypic functions
- *or* generic functions
- *or* type-indexed functions/values

in literature.

Generic H∀SKELL is about adding the possibility to define own generic functions to the Haskell language.

Structure types

In order to make the notion of "structure" of types more concrete, we first have to move to a more abstract view on types.

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Tree}\ a & = & \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\
 & \mid & \mathit{Leaf} \\
\mathbf{data}\ \mathit{Tree}^{\circ}\ a & = & (\mathit{Tree}\ a) \times a \times (\mathit{Tree}\ a) \\
 & + & ()
\end{array}
$$

What has happened?

- Constructor names have disappeared.
- Components of a constructor are now connected by a binary $\times$ operator.
- Different constructors are now connected by a binary $+$ operator.
- The empty constructor Leaf has been replaced by a special unit type $()$.

(The mathematical symbols $+$, $\times$ and 1 are chosen for a reason: In category theory types can really be viewed a sums and products.)

A generic definition for equality

$$
\begin{array}{lcl}
equal\langle a\rangle & :: & a \to a \to Bool \\
equal\langle ()\rangle\ ()\ () & = & True \\
equal\langle Int\rangle\ i_1\ i_2 & = & equalInt\ i_1\ i_2 \\
equal\langle Char\rangle\ c_1\ c_2 & = & equalChar\ c_1\ c_2 \\
equal\langle +\rangle\ equal_A\ equal_B\ (Inl\ a_1)\ (Inl\ a_2) & = & equal_A\ a_1\ a_2 \\
equal\langle +\rangle\ equal_A\ equal_B\ (Inr\ b_1)\ (Inr\ b_2) & = & equal_B\ b_1\ b_2 \\
equal\langle +\rangle\ equal_A\ equal_B\ \_\ \_ & = & False \\
equal\langle \times\rangle\ equal_A\ equal_B\ (a_1, b_1)\ (a_2, b_2) & = & equal_A\ a_1\ a_2 \wedge equal_B\ b_1\ b_2
\end{array}
$$

This definition is similar in spirit to the equality definition for trees earlier, and it reflects the general informal principle to first check if both values belong to the same constructor, then to compare the components.

The requirement that we have to know how to compute equality for the components is met by assuming that we get the needed information as extra arguments in the cases $+$ and $\times$.

We assume that we can make use of the builtin equality functions on base types like integers and characters ($equalInt$, $equalChar$).

A new example: *encode*

$$
\begin{array}{lcl}
\mathbf{data}\ Bit & = & \mathtt{O} \mid \mathtt{I} \\
encode\langle a\rangle & :: & a \to [Bit] \\
encode\langle ()\rangle\ () & = & [\,] \\
encode\langle Int\rangle\ i & = & encodeInt\ i \\
encode\langle Char\rangle\ c & = & encodeChar\ c \\
encode\langle +\rangle\ encode_A\ encode_B\ (Inl\ a) & = & \mathtt{O} : encode_A\ a \\
encode\langle +\rangle\ encode_A\ encode_B\ (Inr\ b) & = & \mathtt{I} : encode_B\ b \\
encode\langle \times\rangle\ encode_A\ encode_B\ (a, b) & = & encode_A\ a +\!\!+\ encode_B\ b
\end{array}
$$

The idea of *encode* is to transform a value of an arbitrary datatype into a sequence of bits. It is thus similar to *show*, but it does not create a string representation intended for user interaction, but rather a binary representation suitable for storage.

The positions of the constructors are encoded. Thus the way in which **data** declarations are transformed can affect the result.

About map and $show$

- map
  The function map is just the generic identity function specialised to type constructors. We will learn more about that in PART II.
- $show$
  To implement $show$ we need to access the constructor names. Therefore, the simple mapping from Haskell types to structure types that ignores the constructors is not suitable. But there is no problem to include constructor names in the mapping. (We will not do that here to simplify the presentation.)

PART II

Generic H∀SKELL compiler

Currently, the compiler for generic function definitions is intended to be a preprocessor for Haskell:

Input: A **Generic H∀SKELL** source is given in Haskell extended with syntax to write down generic definitions.

Output: A Haskell file that can be unsterstood by an ordinary Haskell compiler/interpreter.

Since generic functions depend on *types* and type information is known at *compile time*, it is possible to determine for which types a generic function is used/called in a program.

Therefore the task of the **Generic H∀SKELL** compiler is to generate *specialised* versions of the generic functions for the types at which they are actually used.

Types possess kinds

Let's discuss the shape of type definitions in Haskell again by looking at three examples:

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Bit} & = & \mathtt{I} \mid \mathtt{O} \\
\mathbf{data}\ \mathit{BitList} & = & \mathit{Cons}\ \mathit{Bit}\ \mathit{BitList} \\
 & \mid & \mathit{Nil} \\
\mathbf{data}\ \mathit{Tree}\ a & = & \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\
 & \mid & \mathit{Leaf}
\end{array}
$$

Types can contain recursion, abstraction and application. Types can be classified with kinds (types of types).

$$
\begin{array}{lcl}
\mathit{Bit} & :: & \star \\
\mathit{BitList} & :: & \star \\
\mathit{Tree} & :: & \star \to \star
\end{array}
$$

Kinds of base types

The operations and constants occuring in structure types can themselves be seen as types:

$$\begin{array}{lcl}\mathbf{data}\ () & = & () \\ \mathbf{data}\ a + b & = & \mathit{Inl}\ a \mid \mathit{Inr}\ b \\ \mathbf{data}\ a \times b & = & (a, b)\end{array}$$

$$\begin{array}{ccl}() & :: & \star \\ + & :: & \star \to \star \to \star \\ \times & :: & \star \to \star \to \star\end{array}$$

Type-indexed values possess kind-indexed types

Generic functions can be specialised to types of arbitrary kind, but the type of the function is dependent on the kind of the type argument.

Remember equality: We can define equality on lists if we know equality on the argument type.

$$
\begin{array}{lcl}
[\,] & :: & \star \to \star \\
equal\langle[\,]\rangle & :: & (a \to a \to Bool) \to ([\,a\,] \to [\,a\,] \to Bool)
\end{array}
$$

In general:

$$
\begin{array}{lcl}
equal\langle t :: \kappa\rangle & :: & Equal\langle\kappa\rangle\ t \\
\mathbf{poly}\ Equal\langle\star\rangle\ t & = & t \to t \to Bool \\
\mathbf{poly}\ Equal\langle\rho \to \tau\rangle & = & \forall\ a.Equal\langle\rho\rangle\ a \to Equal\langle\tau\rangle\ (t\ \ a)
\end{array}
$$

Now: Replace *type abstraction* and *type application* by *value abstraction* and *value application*.

Specialisation

Let us specialise the equality function to Bit. We define a "structure type" for Bit.

$$\begin{aligned} \mathbf{data}\ Bit &= \mathtt{O} \mid \mathtt{I} \\ \mathbf{data}\ Bit^{\circ} &= () + () \end{aligned}$$

Now it is easy to specialise equality to Bit°.

$$\begin{aligned} equalBitS &:: Bit^{\circ} \to Bit^{\circ} \to Bool \\ equalBitS &= equal\langle + \rangle\ equal\langle () \rangle\ equal\langle () \rangle \end{aligned}$$

But we still have to provide a wrapper that maps the Bit to Bit°!

Another example: $Tree$

The type $Tree$ is of kind $\star \to \star$.

$$
\begin{array}{lcl}
\mathbf{data}\ Tree\ a & = & Node\ (Tree\ a)\ a\ (Tree\ a) \\
 & \mid & Leaf \\
\mathbf{data}\ Tree^{\circ}\ a & = & (Tree\ a) \times (a \times (Tree\ a)) \\
 & + & () \\
equalTreeS & :: & (a \to a \to Bool) \to Tree^{\circ} \to Tree^{\circ} \to Bool \\
equalTreeS\ equal_A & = & equal\langle + \rangle \\
 & & \quad (equal\langle \times \rangle\ (equalTree\ equal_A) \\
 & & \qquad\qquad (equal\langle \times \rangle\ equal_A\ (equalTree\ equal_A)) \\
 & & \quad) \\
 & & \quad equal\langle () \rangle
\end{array}
$$

Again, a wrapper $equalTree$ that maps $Tree$ to $Tree^{\circ}$ is still missing.

Task summary

The **Generic H∀SKELL** compiler has to:

- parse **data** declarations
- determine the kind of data types
- generate structure types
- parse generic function definitions (with kind-indexed types)
- generate a list of specialisation requests
- generate specialisations of the functions to the structure types
- generate wrappers

Parsing

- Written in Haskell.
- Makes use of the `UU_Parsing` parser combinator library by Doaitse Swierstra.
- Parses most standard Haskell syntax with a few exceptions:
  - No module import/export lists.
  - No classes!
  - No **infix** declarations.
- The goal is to remove these restrictions in the future.
- On the other hand, there are some extensions to the syntax:
  - A **poly** construct to define kind-indexed types.
  - $\langle\rangle$ notation to define type-indexed values.
  - A more liberal syntax for types (explicit $\forall$).

Kind inferencing

- Written in Haskell.
- Similar to type inferencing, but simpler.
- There are no “polymorphic” kinds in Haskell.
- The data is needed to direct the specialisation process.

Type inferencing/checking?

There is no type inferencing or type checking at the moment.

- It is still a partially open problem if it is possible to infer polykinded types.
- Type checking of generic functions is possible if a polykinded type is present. All components of a generic function definition must have the polykinded type of the function, instantiated to the kind of the type constructor.
- Type signatures are generated for all components in the Haskell output, so the Haskell compiler will do type checking for us (but the error messages are not very comprehensible).
- Polykinded types must fulfill additional theoretical requirements. These are not checked.

Structure types and conversion functions

- For each datatype in the input file, one structure type in the output file is generated.
- In addition, two mappings are generated (called an embedding-projection pair) to map a value from the original type to a value of the structure type and vice versa.

We have seen this before:

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Tree}\ a & = & \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\
 & \mid & \mathit{Leaf} \\
\mathbf{data}\ \mathit{Tree}^{\circ}\ a & = & \mathit{SUM}\ (\mathit{PROD}\ (\mathit{Tree}\ a)\ (\mathit{PROD}\ a\ (\mathit{Tree}\ a))) \\
 & & \qquad \mathit{UNIT}
\end{array}
$$

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{EP}\ a\ b & = & \mathit{EP}\{\mathit{from} :: a \to b, \mathit{to} :: b \to a\} \\
\mathit{epTree} & :: & \mathit{EP}\ (\mathit{Tree}\ a)\ (\mathit{Tree}^{\circ}\ a) \\
\mathit{epTree} & = & \mathit{EP}\ \mathit{from}\ \mathit{to} \\
\quad \mathbf{where} & & \\
\quad \mathit{from}\ (\mathit{Node}\ a\ b\ c) & = & \mathit{Inl}\ (a, (b, c)) \\
\quad \mathit{from}\ \mathit{Leaf} & = & \mathit{Inr}\ () \\
\quad \mathit{to}\ (\mathit{Inl}\ (a, (b, c))) & = & \mathit{Node}\ a\ b\ c \\
\quad \mathit{to}\ (\mathit{Inr}\ ()) & = & \mathit{Leaf}
\end{array}
$$

Transforming generic functions

- For each component of a generic function, one function in the output file (with type signature) is generated.

$$
\begin{array}{lcl}
encode\langle()\rangle\ () & = & [\,] \\
encode\langle Int\rangle\ i & = & encodeInt\ i \\
encode\langle Char\rangle\ c & = & encodeChar\ c \\
encode\langle+\rangle\ encode_A\ encode_B\ (Inl\ a) & = & \mathtt{0} : encode_A\ a \\
encode\langle+\rangle\ encode_A\ encode_B\ (Inr\ b) & = & \mathtt{I} : encode_B\ b \\
encode\langle\times\rangle\ encode_A\ encode_B\ (a, b) & = & encode_A\ a +\!\!+\ encode_B\ b
\end{array}
$$

is transformed into

$$
\begin{array}{lcl}
encodeUNIT & :: & UNIT \to [Bit] \\
encodeUNIT\ () & = & [\,] \\
encodeInt & :: & Int \to [Bit] \\
encodeInt\ i & = & encodeInt\ i \\
encodeChar & :: & Char \to [Bit] \\
encodeChar\ c & = & encodeChar\ c \\
encodeSUM & :: & (a \to [Bit]) \to (b \to [Bit]) \to SUM\ a\ b \to [Bit] \\
encodeSUM\ encode_A\ encode_B\ (Inl\ a) & = & \mathtt{0} : encode_A\ a \\
encodeSUM\ encode_A\ encode_B\ (Inr\ b) & = & \mathtt{I} : encode_B\ b \\
encodePROD & :: & (a \to [Bit]) \to (b \to [Bit]) \to PROD\ a\ b \to [Bit] \\
encodePROD\ encode_A\ encode_B\ (a, b) & = & encode_A\ a +\!\!+\ encode_B\ b
\end{array}
$$

Specialisation requests

- For all generic functions in the source file and for all datatype declarations in the source file, a specialisation of that function to that datatype is generated.
- A specialisation consists of one function that operates on structure types (as we have seen before) and one wrapper function that operates on "real" types.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Tree}\ a & = & \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a) \\
 & \mid & \mathit{Leaf} \\
\textbf{data}\ \mathit{Tree}^{\circ}\ a & = & \mathit{SUM}\ (\mathit{PROD}\ (\mathit{PROD}\ (\mathit{Tree}\ a)\ a)\ (\mathit{Tree}\ a)) \\
 & & \qquad\quad \mathit{UNIT} \\
\mathit{encodeTree}^{\circ} & :: & (a \rightarrow [\mathit{Bit}]) \rightarrow \mathit{Tree}^{\circ} \rightarrow [\mathit{Bit}] \\
\mathit{encodeTree}^{\circ}\ \mathit{encode}_A & = & \mathit{encodeSUM} \\
 & & \quad (\mathit{encodePROD}\ (\mathit{encodeTree}\ \mathit{encode}_A) \\
 & & \qquad\qquad\qquad\qquad (\mathit{encodePROD}\ \mathit{encode}_A\ (\mathit{encodeTree}\ \mathit{encode}_A)) \\
 & & \quad) \\
 & & \quad \mathit{encodeUNIT} \\
\mathit{encodeTree} & :: & (a \rightarrow [\mathit{Bit}]) \rightarrow \mathit{Tree} \rightarrow [\mathit{Bit}] \\
\mathit{encodeTree}\ \mathit{encode}_A & = & \mathit{to}\ (\mathit{bimapEncode}\ \mathit{epTree})\ (\mathit{encodeTree}^{\circ}\ \mathit{encode}_A)
\end{array}
$$

Bimaps

- The $encodeTree$ function makes use of a function $bimapEncode$. Such bimaps are generated for all kind-indexed types and are embedding-projection-pair-transformers! (We will not go into more detail here.)

$$
\begin{array}{lcl}
bimapEncode & :: & EP\ a\ a^{\circ} \to EP\ (a \to [Bit])\ (a^{\circ} \to [Bit]) \\
bimapEncode\ bimapt & = & (bimapFun\ bimapt)\ (bimapLst\ bimapBit)
\end{array}
$$

Usage of AG

The `UU_AG` system is used for the specialisation process.

- Data from many different parts of the program has to be collected:
  - Which generic functions are there?
  - Which polykinded type do they have?
  - Which datatypes are there to specialise a generic function to?
  - Which kinds do these datatypes have?
- This data is needed again in different places:
  - In the **poly** constructs to instantiate the polykinded types.
  - In the generic function definitions to generate the specialisations.
  - In the calls to generic functions to replace these calls by calls to the generated specialisations.

Using an attribute grammar to synthesise and redistribute this information makes extensions to the compiler easier.

Reminder: Functionality of the AG system

The UU_AG system operates on an abstract syntax tree.

- Information can be computed at certain places by defining *synthesised attributes*.
- Information can be passed around and used in the tree easily by defining *inherited* and *chained attributes*.
- The UU_AG compiler itself transforms an attribute grammar input file into a Haskell output file. The attribute definitions are translated into catamorphisms on the abstract syntax tree.

Shortcomings of AG

Generic H∀SKELL in its current form differs from classical compilers in that it does generate code that is not much different from its input language.

Many parts are left unchanged. Most of the work is best expressed as a transformation on a slightly extended Haskell language. Such a transformation is hard to express in an AG because a computed replacement for a language construct cannot really "replace" the original construct (i.e. one cannot define attributes for attributes).

Higher-order attributes address this problem by allowing exactly this and might be implemented in a future version of the AG system.

Future work

- Include the missing parts of Haskell into the parser.
  - Add module support.
- Write a language report.
- Add typechecking?
- Add more generic programming functionality:
  - Other styles of generic function definitions.
  - Deal with type classes.
  - Do generic programming on the type level: type-indexed types.

Links

If you want to learn more about Generic Programming, have a look at the following web pages:

- The **Generic H∀SKELL** project page:
  `http://www.generic-haskell.org`
- Ralf Hinze's publications page:
  `http://www.cs.uu.nl/~ralf/publications.html`
- The polytypic programming page, maintained by Patrik Jansson:
  `http://www.cs.chalmers.se/~patrikj/poly`
  (This page contains information about PolyP, a predecessor of **Generic H∀SKELL**.)
- The Glasgow Haskell compiler:
  `http://haskell.org/ghc`
  (A possibility to define generic functions in type classes is implemented in the new version 5.00.)