Generic H∀SKELL, Specifically

Dave Clarke and Andres Löh

July 11, 2002

Introduction

This talk is about recent extensions to the Generic H∀SKELL language.

These extensions have in common that they

- are motivated by examples that we could not do nicely before,
- allow to program a larger class of generic programs with less work,
- extend the application area of Generic H∀SKELL to generic traversals over a large tree (of values of different datatypes).

Structure

- Plain Generic H∀SKELL
- Specific Generic H∀SKELL
  - Default cases
  - Constructor cases
- Conclusions

PART I

Plain Generic H∀SKELL

Generic H∀SKELL provides . . .

- a superset of Haskell 98 (which is compiled to Haskell)
- constructs for defining generic (i.e. type-indexed) functions
- generic programming in a style based on work by Hinze

Type-indexed values have kind-indexed types

Example: *gmap*

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Conference} & = & \mathit{MPC} \mid \mathit{WCGP} \\
\textbf{data}\ \mathit{List}\ a & = & \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \\
\textbf{data}\ \mathit{Tree}\ a\ b & = & \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{Tree}\ a\ b)\ b\ (\mathit{Tree}\ a\ b)
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{gmap}\{\!|\mathit{Conference}|\!\}\ \mathit{MPC} & \rightsquigarrow & \mathit{MPC} \\
\mathit{gmap}\{\!|\mathit{List}|\!\}\ (+1)\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil})) & \rightsquigarrow & \mathit{Cons}\ 3\ (\mathit{Cons}\ 4\ \mathit{Nil}) \\
\mathit{gmap}\{\!|\mathit{Tree}|\!\}\ (*2)\ (\mathit{const}\ \texttt{"bar"})\ (\mathit{Node}\ (\mathit{Leaf}\ 21)\ \texttt{"foo"}\ (\mathit{Leaf}\ 5)) & & \\
\qquad \rightsquigarrow & \mathit{Node}\ (\mathit{Leaf}\ 42)\ \texttt{"bar"}\ (\mathit{Leaf}\ 10)
\end{array}
$$

Type-indexed values have kind-indexed types – continued

$$
\begin{array}{lll}
gmap\{\!|\, Conference :: * |\!\} & :: & Conference \to Conference \\
gmap\{\!|\, List :: * \to * |\!\} & :: & \forall\ a\ b\ .\ (a \to b) \to List\ a \to List\ b \\
gmap\{\!|\, Tree :: * \to * \to * |\!\} & :: & \\
\quad \forall\ a\ b\ c\ d\ .\ (a \to c) \to (b \to d) \to Tree\ a\ b \to Tree\ c\ d &&
\end{array}
$$

The following shows a valid Generic H∀SKELL definition of a kind-indexed type and the assignment of this type to the generic function $gmap$:

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

Type-indexed values have kind-indexed types – continued

The function $gmap$ can be defined as

$$
\begin{array}{lllcl}
gmap\{\!|t :: k|\!\} & & & :: & Map\{\!|k|\!\}\ t\ t \\
gmap\{\!|Unit|\!\} & & Unit & = & Unit \\
gmap\{\!|:+:|\!\} & gmapA\ gmapB & (Inl\ a) & = & Inl\ (gmapA\ a) \\
gmap\{\!|:+:|\!\} & gmapA\ gmapB & (Inr\ b) & = & Inr\ (gmapB\ b) \\
gmap\{\!|:*:|\!\} & gmapA\ gmapB & (a :*: b) & = & (gmapA\ a) :*: (gmapB\ b) \\
gmap\{\!|Con\ c|\!\} & gmapA & (Con\ a) & = & Con\ (gmapA\ a) \\
gmap\{\!|Label\ l|\!\} & gmapA & (Label\ a) & = & Label\ (gmapA\ a) \\
gmap\{\!|Int|\!\} & & i & = & i \\
gmap\{\!|Char|\!\} & & c & = & c
\end{array}
$$

Observations:

- Type arguments can be type constructors of arbitrary kind.
- There are special constructors (Con, $:+:$) to reflect the *structure of types* as well as abstract types (Int, $Char$).
- Each case matches the generic function's type.

Compilation of generic functions

- Each case is converted into one Haskell function.
- The type of each case is determined by the function's type.

$$
\begin{array}{l}
\vdots \\
case\_gmap\_Sum \;::\; \forall\; a\; b\; c\; d\;.\; (a \to c) \to (b \to d) \to a\,{:}{+}{:}\,b \to c\,{:}{+}{:}\,d \\
case\_gmap\_Sum\;\; gmapA\; gmapB\; (Inl\; a) \;=\; Inl\; (gmapA\; a) \\
case\_gmap\_Sum\;\; gmapA\; gmapB\; (Inr\; b) \;=\; Inr\; (gmapB\; b) \\
case\_gmap\_Prod \;::\; \forall\; a\; b\; c\; d\;.\; (a \to c) \to (b \to d) \to a\,{:}{*}{:}\,b \to c\,{:}{*}{:}\,d \\
case\_gmap\_Prod\;\; gmapA\; gmapB\; (a{:}{*}{:}b) \;=\; (gmapA\; a){:}{*}{:}(gmapB\; b) \\
\vdots
\end{array}
$$

Structure of data types

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

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

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

Structure of data types – continued

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

- For each datatype in a program, Generic H∀SKELL generates an equivalent data type using `:+:`, `:*:`, $Unit$, Con and $Label$.
- An embedding-projection pair mapping between both versions of the type is also generated.

$$
\begin{array}{lcl}
\textbf{data}\ EP\ a\ b & = & EP\{from :: a \to b, to :: b \to a\} \\
ep\_List & :: & EP\ (List\ a)\ (List^{\circ}\ a)
\end{array}
$$

Structure of datatypes – more examples

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

Specialisation of generic functions – example

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

$$
\begin{array}{lll}
\mathit{gmap}\{\!|\,\mathit{Tree}\ \mathit{String}\,|\!\} & \rightsquigarrow & \mathit{gmap}\{\!|\,\mathit{Tree}\,|\!\}\ (\mathit{gmap}\{\!|\,\mathit{String}\,|\!\}) \\
\mathit{gmap}\{\!|\,\mathit{Tree}\,|\!\} & \rightsquigarrow & \text{wrapper around } \mathit{gmap}\{\!|\,\mathit{Tree}^{\circ}\,|\!\} \\
\mathit{gmap}\{\!|\,\mathit{Tree}^{\circ}\,|\!\} & \rightsquigarrow & \lambda a\ b \longrightarrow \mathit{gmap}\{\!|\,\texttt{:+:}\,|\!\} \\
 & & \qquad (\mathit{gmap}\{\!|\,\mathit{Con}\,|\!\}\ a) \\
 & & \qquad (\mathit{gmap}\{\!|\,\texttt{:*:}\,|\!\} \\
 & & \qquad\quad (\mathit{gmap}\{\!|\,\mathit{Tree}\,|\!\}\ a\ b) \\
 & & \qquad\quad (\mathit{gmap}\{\!|\,\texttt{:*:}\,|\!\} \\
 & & \qquad\qquad b\ (\mathit{gmap}\{\!|\,\mathit{Tree}\,|\!\}\ a\ b) \\
 & & \qquad\quad) \\
 & & \qquad) \\
\mathit{gmap}\{\!|\,\mathit{String}\,|\!\} \rightsquigarrow \mathit{gmap}\{\!|\,[\,\mathit{Char}\,]\,|\!\} & \rightsquigarrow & \mathit{gmap}\{\!|\,[\,]\,|\!\}\ (\mathit{gmap}\{\!|\,\mathit{Char}\,|\!\}) \\
\mathit{gmap}\{\!|\,[\,]\,|\!\} & \rightsquigarrow & \text{wrapper around } \mathit{gmap}\{\!|\,[\,]^{\circ}\,|\!\} \\
\mathit{gmap}\{\!|\,[\,]^{\circ}\,|\!\} & \rightsquigarrow & \ldots
\end{array}
$$

Specialisation of generic functions – type expressions

$poly\{|\ UltraComplexTree\ (Mostly\ Harmless)\ ElementType\ |\}$

- Complex type expressions and type synonyms in type arguments are simplified: type application is replaced by value application, type abstraction is replaced by type abstraction

$poly\{|\ UltraComplexTree\ |\}\ (poly\{|\ Mostly\ |\}\ (poly\{|\ Harmless\ |\}))$
$(poly\{|\ ElementType\ |\})$

Specialisation of generic functions – named types

$poly\{\!|\,Mostly\,|\!\}$
$poly\{\!|\,Harmless\,|\!\}$

There are two possibilities how to deal with simple type arguments:

- If there is a case for that type in the generic function, then the specialisation equals that case.

$$\begin{array}{lcl} poly\{\!|\,Harmless\,|\!\} & \rightsquigarrow & spec\_poly\_Harmless \\ spec\_poly\_Harmless & = & case\_poly\_Harmless \end{array}$$

- Otherwise, the default generic behaviour takes place.

$$\begin{array}{lcl} poly\{\!|\,Mostly\,|\!\} & \rightsquigarrow & spec\_poly\_Mostly \\ spec\_poly\_Mostly & \rightsquigarrow & \text{wrapper around } poly\{\!|\,Mostly^{\circ}\,|\!\} \end{array}$$

PART II

New features of Generic H∀SKELL

Example datatypes

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Var} & = & \mathit{V}\ \mathit{String} \\
\textbf{data}\ \mathit{Type} & = & \mathit{TVar}\ \mathit{Var} \\
 & | & \mathit{Arrow}\ \mathit{Type}\ \mathit{Type} \\
\textbf{data}\ \mathit{Expr} & = & \mathit{Var}\ \mathit{Var} \\
 & | & \mathit{App}\ \mathit{Expr}\ \mathit{Expr} \\
 & | & \mathit{Lambda}\ (\mathit{Var}, \mathit{Type})\ \mathit{Expr} \\
 & | & \mathit{Let}\ (\mathit{Var}, \mathit{Type})\ \mathit{Expr}\ \mathit{Expr}
\end{array}
$$

Default cases

There is a large class of generic functions that do almost nothing (pass on results, preserve structure, . . .) in a large part of the tree.

But in a few places something special happens.

```
-- type VarCollect{[*]} t                          =  t → [Var]
varcollect{[t :: k]}                               ::  VarCollect{[k]} t
varcollect{[Unit]}                        x        =  []
varcollect{[:+:]}                cA cB (Inl a)     =  cA a
varcollect{[:+:]}                cA cB (Inr b)     =  cB b
varcollect{[:*:]}                cA cB (a :*: b)   =  cA a `union` cB b
varcollect{[Con c]}              cA    (Con a)     =  cA a
varcollect{[Label l]}            cA    (Label a)   =  cA a
varcollect{[Char]}                        x        =  []
varcollect{[Int]}                         x        =  []
varcollect{[Var]}                         v        =  [v]
```

Only the last line (and maybe the product case) is really application specific. There are many collection functions over other data types that share all the lines except the Var line.

Default cases – continued

Default cases allow to abstract out common cases from generic functions.

We start by defining a very general function which is not useful in itself:

$$
\begin{array}{lllll}
\textbf{type}\ \mathit{Collect}\{\![*]\!\}\ & t\ a & = & t \to [a] \\
\textbf{type}\ \mathit{Collect}\{\![k \to l]\!\}\ & t\ a & = & \\
\quad \forall\ u\ .\ \mathit{Collect}\{\![k]\!\}\ u\ a \to \mathit{Collect}\{\![l]\!\}\ (t\ u)\ a & & & \\
\mathit{collect}\{\!|\ t :: k\ |\!\} & & :: & \forall\ a\ .\ \mathit{Collect}\{\![k]\!\}\ t\ a \\
\mathit{collect}\{\!|\ \mathit{Unit}\ |\!\} & x & = & [\,] \\
\mathit{collect}\{\!|\ \texttt{:+:}\ |\!\} & cA\ cB\ (\mathit{Inl}\ a) & = & cA\ a \\
\mathit{collect}\{\!|\ \texttt{:+:}\ |\!\} & cA\ cB\ (\mathit{Inr}\ b) & = & cB\ b \\
\mathit{collect}\{\!|\ \texttt{:*:}\ |\!\} & cA\ cB\ (a\ \texttt{:*:}\ b) & = & cA\ a +\!\!+ cB\ b \\
\mathit{collect}\{\!|\ \mathit{Con}\ c\ |\!\} & cA\ (\mathit{Con}\ a) & = & cA\ a \\
\mathit{collect}\{\!|\ \mathit{Label}\ l\ |\!\} & cA\ (\mathit{Label}\ a) & = & cA\ a \\
\mathit{collect}\{\!|\ \mathit{Char}\ |\!\} & x & = & [\,] \\
\mathit{collect}\{\!|\ \mathit{Int}\ |\!\} & x & = & [\,]
\end{array}
$$

Default cases – continued

Now $varcollect$ can be written as *extension* or *modification* of $collect$:

$$
\begin{array}{lll}
varcollect\{\!|\, t :: k \,|\!\} & :: & Collect\{\![k]\!\}\ t\ Var \\
varcollect\{\!|\, Var \,|\!\}\ v & = & [v] \\
varcollect\{\!|\, :*: \,|\!\}\ cA\ cB\ (a :*: b) & = & cA\ a\ `union`\ cB\ b \\
varcollect\{\!|\, c \,|\!\} & = & collect\{\!|\, c \,|\!\}
\end{array}
$$

- The last line is the default case.
- Default cases always have exactly this form.
- Order of the cases is irrelevant (unlike Haskell pattern matching).
- All cases not given explicitly are "inherited" from $collect$.
- Cases can be added (Var) or redefined (`:*:`).

Default cases – translation

$$
\begin{array}{lll}
varcollect\{\!|\, t :: k \,|\!\}\ \ \ \ \ \ \ \ \ \ \ \ & :: & Collect\{\![k]\!\}\ t\ Var \\
varcollect\{\!|\, Var \,|\!\}\ \ \ \ \ \ \ \ v & = & [\, v \,] \\
varcollect\{\!|\, :*: \,|\!\}\ \ \ cA\ cB\ (a\ :*:\ b) & = & cA\ a\ `union`\ cB\ b \\
varcollect\{\!|\, Unit \,|\!\} & = & collect\{\!|\, Unit \,|\!\} \\
varcollect\{\!|\, :+: \,|\!\} & = & collect\{\!|\, Unit \,|\!\} \\
varcollect\{\!|\, Con\ c \,|\!\} & = & collect\{\!|\, Con\ c \,|\!\} \\
varcollect\{\!|\, Label\ l \,|\!\} & = & collect\{\!|\, Label\ l \,|\!\} \\
varcollect\{\!|\, Char \,|\!\} & = & collect\{\!|\, Char \,|\!\} \\
varcollect\{\!|\, Int \,|\!\} & = & collect\{\!|\, Int \,|\!\}
\end{array}
$$

This is the same as . . .

Default cases – translation, continued

. . . the original definition:

$$
\begin{array}{lllcl}
varcollect\{\!|\, t :: k \,|\!\}\ & & & :: & Collect\{\!|\, k \,|\!\}\ t\ Var \\
varcollect\{\!|\, Var \,|\!\}\ & & v & = & [\, v \,] \\
varcollect\{\!|\, :*: \,|\!\}\ & cA\ cB & (a :*: b) & = & cA\ a\ `union`\ cB\ b \\
varcollect\{\!|\, Unit \,|\!\}\ & & x & = & [\,] \\
varcollect\{\!|\, :+: \,|\!\}\ & cA\ cB & (Inl\ a) & = & cA\ a \\
varcollect\{\!|\, :+: \,|\!\}\ & cA\ cB & (Inr\ b) & = & cB\ b \\
varcollect\{\!|\, Con\ c \,|\!\}\ & cA & (Con\ a) & = & cA\ a \\
varcollect\{\!|\, Label\ l \,|\!\}\ & cA & (Label\ a) & = & cA\ a \\
varcollect\{\!|\, Char \,|\!\}\ & & x & = & [\,] \\
varcollect\{\!|\, Int \,|\!\}\ & & x & = & [\,]
\end{array}
$$

Note that *varcollect* does not contain any recursive calls to *collect*! All *Var* occurrences, no matter how deep they are in the tree, will be collected by *varcollect*.

Extending a generic function by means of a default case does modify the original behaviour in a deep fashion.

Constructor cases

- Generic H∀SKELL allows special cases for specific types, but sometimes that is not fine-grained enough!
- Datatypes can have a large number of constructors, and often only one or two constructors need special treatment, whereas the others follow the generic behaviour.
- With a special case for the datatype, the generic behaviour for the uninteresting constructors would need to be reprogrammed explicitly.

$\Longrightarrow$ Use constructor cases!

Collecting free variables

Once more the example datatypes:

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Var} & = & \mathit{V}\ \mathit{String} \\
\textbf{data}\ \mathit{Type} & = & \mathit{TVar}\ \mathit{Var} \\
 & | & \mathit{Arrow}\ \mathit{Type}\ \mathit{Type} \\
\textbf{data}\ \mathit{Expr} & = & \mathit{Var}\ \mathit{Var} \\
 & | & \mathit{App}\ \mathit{Expr}\ \mathit{Expr} \\
 & | & \mathit{Lambda}\ (\mathit{Var}, \mathit{Type})\ \mathit{Expr} \\
 & | & \mathit{Let}\ (\mathit{Var}, \mathit{Type})\ \mathit{Expr}\ \mathit{Expr}
\end{array}
$$

Let us collect the free variables now:

- In principle the same as $\mathit{varcollect}$.
- For Lambda and Let, special actions are required.

Collecting free variables – continued

$$
\begin{array}{lll}
\mathit{freecollect}\{\!|\, t :: k \,|\!\} & :: & \mathit{Collect}\{\!|k|\!\}\ t\ \mathit{Var} \\
\mathit{freecollect}\{\!|\, c \,|\!\} & = & \mathit{varcollect}\{\!|\, c \,|\!\} \\
\mathit{freecollect}\{\!|\ \mathbf{case}\ \mathit{Lambda}\ |\!\}\ (\mathit{Lambda}\ (v,t)\ e) & = & \mathit{filter}\ (\not\equiv v)\ (\mathit{freecollect}\{\!|\mathit{Expr}|\!\}\ e) \\
\mathit{freecollect}\{\!|\ \mathbf{case}\ \mathit{Let}\ |\!\}\ \ (\mathit{Let}\ (v,t)\ e\ e') & = & \mathit{freecollect}\{\!|\mathit{Expr}|\!\}\ e \\
 & & +\!\!+\ \mathit{filter}\ (\not\equiv v)\ (\mathit{freecollect}\{\!|\mathit{Expr}|\!\}\ e')
\end{array}
$$

In the presence of constructor cases, the type arguments in the definitions may have the following forms:

- a named type constructor,
- the special type constructors Con or $Label$ followed by a variable name,
- a single variable (for default cases),
- the keyword **case** followed by the name of a constructor.

More about constructor cases

Cases for constructors have the same type as cases for the corresponding type (would) have.

$$
\begin{array}{lcl}
\text{-- } Lambda & :: & (Var, Type) \to Expr \to Expr \\
\text{-- } freecollect\{\!|\ \mathbf{case}\ Lambda\ |\!\}\ (Lambda\ (v,t)\ e) & :: & Collect\{\!|*|\!\}\ t\ Expr \\
freecollect\{\!|\ \mathbf{case}\ Lambda\ |\!\}\quad (Lambda\ (v,t)\ e) & = & filter\ (\not\equiv v)\ (freecollect\{\!|Expr|\!\}\ e) \\
case\_freecollect\_Case\_Lambda & :: & Expr \to [\,Var\,] \\
case\_freecollect\_Case\_Lambda\ (Lambda\ (v,t)\ e) & = & filter\ (\not\equiv v)\ (freecollect\{\!|Expr|\!\}\ e)
\end{array}
$$

Generic functions containing constructor cases are still *generic*. For instance, the call

$$freecollect\{\!|\ Tree\ Expr\ Int|\!\}$$

would collect all variables that appear free in one of the expressions in the tree.

Calling constructor cases

For constructor cases to be actually executed, the type translation has to be adapted:
Before:

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

Now:

$$\begin{array}{lcl} \textbf{data}\ \mathit{List}^{\circ}\ a & = & \mathit{Con}\ (\mathit{Case\_Nil}\ a)\ \texttt{:+:}\ \mathit{Con}\ (\mathit{Case\_Cons}\ a) \\ \textbf{type}\ \mathit{Case\_Nil}\ a & = & \mathit{List}\ a \\ \textbf{type}\ \mathit{Case\_Cons}\ a & = & \mathit{List}\ a \\ \textbf{type}\ \mathit{Case\_Nil}^{\circ}\ a & = & \mathit{Unit} \\ \textbf{type}\ \mathit{Case\_Cons}^{\circ}\ a & = & \mathit{Label}\ a\ \texttt{:*:}\ \mathit{Label}\ (\mathit{List}\ a) \end{array}$$

The specialisation mechanism will do the rest.

Conclusions and future work

- Default and constructor cases allow generic traversals to be conveniently expressed within Generic H∀SKELL.
- Even generic functions can benefit from specific behaviour.
- The extensions are rather ad-hoc, but useful.
- There are still things we can't do yet.

Try and see yourself!

- Generic H∀SKELL 1.23 (Beryl) has been released on July 5 and implements the extensions presented here.
- It is available for download at

  http://www.generic-haskell.org

- There will be a demonstration after lunch.