Generic views on data types

Alexey Rodriguez

Joint work with Stefan Holdermans, Johan Jeuring and Andres Löh

July, 2006

Introduction

- There are several approaches to generic programming: PolyP, Generic Haskell, Scrap Your Boilerplate, etc.
- This talk develops a framework in which a number of these approaches can be encoded in Generic Haskell.
- The framework
  - makes generic programming more expressive: we can implement functions we couldn't implement before in a single framework;
  - makes the definitions of some generic functions easier or more efficient.

Collecting values generically

Consider the following data types defining trees and lambda terms

$$\begin{aligned}&\textbf{data}\ \mathsf{Tree}\ \ \mathsf{a} \quad = \mathit{Leaf} \mid \mathit{Branch}\ (\mathsf{Tree\ a})\ \mathsf{a}\ (\mathsf{Tree\ a})\\ &\textbf{data}\ \mathsf{Term\ var} = \mathit{Var}\ \mathsf{var} \mid \mathit{Abs}\ \mathsf{var\ Term} \mid \mathit{App}\ \mathsf{Term\ Term}\end{aligned}$$

and functions that collect tree elements and term variables respectively

$$\begin{aligned}&\mathit{col}_{\mathsf{Tree}} :: \forall \mathsf{a}\,.\,\mathsf{Tree\ a} \to [\mathsf{a}]\\ &\mathit{col}_{\mathsf{Tree}}\ \mathit{Leaf} = [\,]\\ &\mathit{col}_{\mathsf{Tree}}\ (\mathit{Branch}\ t_1\ x\ t_2) = \mathit{col}_{\mathsf{Tree}}\ t_1 +\!\!+ [x] +\!\!+ \mathit{col}_{\mathsf{Tree}}\ t_2\\ &\mathit{col}_{\mathsf{Term}} :: \forall \mathsf{var}\,.\,\mathsf{Term\ var} \to [\mathsf{var}]\\ &\mathit{col}_{\mathsf{Term}}\ (\mathit{Var}\ x) = [x]\\ &\mathit{col}_{\mathsf{Term}}\ (\mathit{Abs}\ x\ t) = [x] +\!\!+ \mathit{col}_{\mathsf{Term}}\ t\\ &\mathit{col}_{\mathsf{Term}}\ (\mathit{App}\ t\ u) = \mathit{col}_{\mathsf{Term}}\ t +\!\!+ \mathit{col}_{\mathsf{Term}}\ u\end{aligned}$$

As we will see later, we can write a generic function *col* that subsumes these two.

Problem: generic recursive children

Let us turn our attention to functions that return the recursive children of a data type.

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Tree\ a} = \mathit{Leaf} \mid \mathit{Branch}\ (\mathsf{Tree\ a})\ \mathsf{a}\ (\mathsf{Tree\ a}) \\
&\textbf{data}\ \mathsf{Term\ var} = \mathit{Var}\ \mathsf{var} \mid \mathit{Abs}\ \mathsf{var\ Term} \mid \mathit{App}\ \mathsf{Term\ Term}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{subtrees} :: \forall \mathsf{a}\ .\ \mathsf{Tree\ a} \to [\mathsf{Tree\ a}] \\
&\mathit{subtrees}\ (\mathit{Leaf}) = [\,] \\
&\mathit{subtrees}\ (\mathit{Branch}\ l\ x\ r) = [l, r]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{subterms} :: \forall \mathsf{var}\ .\ \mathsf{Term\ var} \to [\mathsf{Term\ var}] \\
&\mathit{subterms}\ (\mathit{Var}\ x) = [\,] \\
&\mathit{subterms}\ (\mathit{Abs}\ x\ t) = [t] \\
&\mathit{subterms}\ (\mathit{App}\ t\ u) = [t, u]
\end{aligned}
$$

It is not possible to write a generic function that returns the recursive children of a data type.

More problems

- Inefficient generic encoding function.
- Some XPath query like functions benefit from a regular representation of data types.
- Until now we could not encode the programming idioms of the SYB approach.

Solution: Generic views on data types

- Generic Haskell has a unique standard representation of data types, thus, generic functions *view* data types in a unique way.
- To solve these problems we extend Generic Haskell with support for additional representations of data types. We call these representations *Generic Views*.
- Creating appropriate generic views, we can write generic functions that solve the problems that we just saw.

Summary

In the rest of this presentation we will look at the following:

- Generic Haskell functions and the standard view,
- the fixed point view and how it solves the recursive children problem,
- the implementation of generic views,
- conclusions.

Structure types in Generic Haskell

In Generic Haskell, a generic function is defined by induction over the structure of types. The *structure types* we use are defined with the following view types:

```
data Unit      = Unit
data Sum  a b = Inl a | Inr b
data Prod a b = a × b
```

A generic collect funtion

We define the type-indexed function *col* to collect values from a data structure.

$$
\begin{array}{lll}
col\ \langle \mathsf{a} :: \star \mid \mathsf{c} :: \star \rangle :: (col\ \langle \mathsf{a} \mid \mathsf{c} \rangle) \Rightarrow \mathsf{a} \to [\mathsf{c}] & & \\
col\ \langle \mathsf{Unit} \rangle & Unit & = [\,] \\
col\ \langle \mathsf{Sum}\ \alpha\ \beta \rangle & (Inl\ a) & = col\ \langle \alpha \rangle\ a \\
col\ \langle \mathsf{Sum}\ \alpha\ \beta \rangle & (Inr\ b) & = col\ \langle \beta \rangle\ b \\
col\ \langle \mathsf{Prod}\ \alpha\ \beta \rangle & (a \times b) & = col\ \langle \alpha \rangle\ a +\!\!+ col\ \langle \beta \rangle\ b \\
col\ \langle \mathsf{Int} \rangle & n & = [\,] \\
col\ \langle \mathsf{Char} \rangle & c & = [\,]
\end{array}
$$

Types for instances:

$$
\begin{array}{l}
col\ \langle \mathsf{Int} \rangle :: \forall \mathsf{c}\,.\,\mathsf{Int} \to [\mathsf{c}] \\
col\ \langle \mathsf{Tree}\ \alpha \rangle :: \forall \mathsf{c}\ \mathsf{a}\,.\,(col\ \langle \alpha \rangle :: \mathsf{a} \to [\mathsf{c}]) \Rightarrow \mathsf{Tree}\ \mathsf{a} \to [\mathsf{c}]
\end{array}
$$

Applying a generic function

To collect the elements of a tree we can use a *generic application*.

$$
\begin{aligned}
&\mathit{tree} :: \mathsf{Tree\ Int} \\
&\mathit{tree} = (\mathit{Branch}\ (\mathit{Branch}\ \mathit{Leaf}\ 1\ \mathit{Leaf})\ 2\ (\mathit{Branch}\ \mathit{Leaf}\ 3\ \mathit{Leaf})) \\
&\mathbf{let}\ \mathit{col}\langle\alpha\rangle = \lambda x \to [x] \\
&\mathbf{in}\ \ \mathit{col}\langle\mathsf{Tree}\ \alpha\rangle\ \mathit{tree} \\
&\quad \rightsquigarrow [1,2,3]
\end{aligned}
$$

The function $\lambda x \to [x]$ in the *local redefinition* specifies what to do with the tree elements.

Collecting variables:

$$
\begin{aligned}
&\mathit{term} :: \mathsf{Term\ String} \\
&\mathit{term} = (\mathit{Abs}\ \texttt{"x"}\ (\mathsf{Var}\ \texttt{"x"})) \\
&\mathbf{let}\ \mathit{col}\langle\alpha\rangle = \lambda x \to [x] \\
&\mathbf{in}\ \ \mathit{col}\langle\mathsf{Term}\ \alpha\rangle\ \mathit{term} \\
&\quad \rightsquigarrow [\texttt{"x"}, \texttt{"x"}]
\end{aligned}
$$

From type-indexed to generic

To obtain an instance of a generic function for a data type that does not appear as type index, we use a structure type.

The structure representation of types is expressed in terms of data types such as units, sums, products, and base types such as integers, characters, etc. For example

$$\begin{aligned}&\textbf{data}\ [\mathsf{a}] \quad = [\,] \mid \mathsf{a} : [\mathsf{a}]\\ &\textbf{data}\ \mathsf{Tree\ a} = \mathit{Leaf} \mid \mathit{Branch}\ (\mathsf{Tree\ a})\ \mathsf{a}\ (\mathsf{Tree\ a})\end{aligned}$$

have the following structure representations:

$$\begin{aligned}&\textbf{type}\ \mathit{Str}([\,])\ \mathsf{a} \quad = \mathsf{Sum\ Unit\ (Prod\ a\ [a])}\\ &\textbf{type}\ \mathit{Str}(\mathsf{Tree})\ \mathsf{a} = \mathsf{Sum\ Unit\ (Prod\ (Tree\ a)\ (Prod\ a\ (Tree\ a))}\end{aligned}$$

Embedding-projection pairs

To move between a type and its isomorphic structure type, we use an *embedding-projection pair*:

$$\textbf{data}\ \mathsf{EP}\ \mathsf{a}\ \mathsf{b} = \mathsf{EP}\{\mathit{from} :: (\mathsf{a} \to \mathsf{b}), \mathit{to} :: (\mathsf{b} \to \mathsf{a})\}$$

A type T and its structural representation type $\mathit{Str}(\mathsf{T})$ are isomorphic, witnessed by a value $\mathit{conv}_{\mathsf{T}} :: \mathsf{EP}\ \mathsf{T}\ \mathit{Str}(\mathsf{T})$. For example, for the list data type we have that $\mathit{conv}_{[\,]} = \mathit{EP}\ \mathit{from}_{[\,]}\ \mathit{to}_{[\,]}$, where $\mathit{from}_{[\,]}$ and $\mathit{to}_{[\,]}$ are defined by

$$\begin{array}{lll}
\mathit{from}_{[\,]} & :: & [\mathsf{a}] \to \mathit{Str}([\,])\ \mathsf{a} \\
\mathit{from}_{[\,]}\ [\,] & = & \mathit{Inl}\ \mathit{Unit} \\
\mathit{from}_{[\,]}\ (a : as) & = & \mathit{Inr}\ (a \times as) \\
\\
\mathit{to}_{[\,]} & :: & \mathit{Str}([\,])\ \mathsf{a} \to [\mathsf{a}] \\
\mathit{to}_{[\,]}\ (\mathit{Inl}\ \mathit{Unit}) & = & [\,] \\
\mathit{to}_{[\,]}\ (\mathit{Inr}\ (a \times as)) & = & a : as
\end{array}$$

Views

- Views were introduced by Wadler to reconcile pattern matching and data abstraction. For example:

$$
\begin{array}{l}
view\ int ::= Zero \mid Succ\ int \\
\quad inn\ n \mid n \equiv 0 \quad = Zero \\
\qquad\quad \mid n > 0 \quad = Succ\ (n-1) \\
\quad out\ Zero \quad\quad = 0 \\
\quad out\ (Succ\ n) = n+1
\end{array}
$$

- The sums and products structure representation of Generic Haskell is only one of several views on data types.

A fixed-point view

Generic recursive children

$$
\begin{aligned}
&\mathit{subtrees} :: \forall \mathsf{a} \,.\, \mathsf{Tree\ a} \to [\mathsf{Tree\ a}] \\
&\mathit{subtrees}\ (\mathit{Leaf}) = [\,] \\
&\mathit{subtrees}\ (\mathit{Branch}\ l\ x\ r) = [l, r]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{subterms} :: \forall \mathsf{var} \,.\, \mathsf{Term\ var} \to [\mathsf{Term\ var}] \\
&\mathit{subterms}\ (\mathit{Var}\ x) = [\,] \\
&\mathit{subterms}\ (\mathit{Abs}\ x\ t) = [t] \\
&\mathit{subterms}\ (\mathit{App}\ t\ u) = [t, u]
\end{aligned}
$$

- Both function *subterms* and *subtrees* calculate the immediate recursive children of a data type.
- Generalizing *subterms* and *subtrees* into a generic function *children* requires access to the recursion.
- Hence, we cannot write *children* in the standard view.

Fixed points

Suppose that the recursive data type Tree

$$\textbf{data}\ \mathsf{Tree\ a} = \mathit{Leaf} \mid \mathit{Branch}\ (\mathsf{Tree\ a})\ \mathsf{a}\ (\mathsf{Tree\ a})$$

is represented by an explicit fixed point of a functor:

$$\begin{aligned}
&\textbf{type}\ \mathit{Str}_F(\mathsf{Tree})\ \mathsf{a} = \mathsf{Fix}\ (\mathsf{TreeBase\ a}) \\
&\textbf{data}\ \mathsf{TreeBase\ a\ r} = \mathit{LeafBase} \\
&\qquad\qquad\qquad\qquad\quad \mid\ \mathit{BranchBase}\ \mathsf{r\ a\ r} \\
&\textbf{data}\ \mathsf{Fix\ f} \qquad\quad = \mathit{In}\ (\mathsf{f}\ (\mathsf{Fix\ f}))
\end{aligned}$$

Now it is possible to have *children* collect the recursive elements in the functor:

$$\begin{aligned}
&\mathit{children}\langle \mathsf{a} :: \star \rangle :: (\forall \mathsf{b}\,.\,\mathit{col}\ \langle \mathsf{a} \mid \mathsf{b} \rangle) \Rightarrow \mathsf{a} \to [\mathsf{a}] \\
&\mathit{children}\langle \mathsf{Fix}\ \phi \rangle\ (\mathit{In}\ r) = \textbf{let}\ \mathit{col}\langle \alpha \rangle\ x = [x]\ \textbf{in}\ \mathit{col}\langle \phi\ \alpha \rangle\ r
\end{aligned}$$

A fixed-point view

- With a *fixed point view*, the compiler automatically derives the fixed point structure type and the corresponding embedding-projection pairs for any recursive data type.

$$
\begin{array}{ll}
\mathit{from}_{\mathsf{Tree}} & :: \mathsf{Tree\ a} \rightarrow \mathit{Str}_F(\mathsf{Tree})\ \mathsf{a} \\
\mathit{from}_{\mathsf{Tree}}\ \mathit{Leaf} & = \mathit{In}\ \mathit{LeafBase} \\
\mathit{from}_{\mathsf{Tree}}\ (\mathit{Branch}\ t_1\ x\ t_2) & = \mathit{In}\ (\mathit{BranchBase}\ (\mathit{from}_{\mathsf{Tree}}\ t_1)\ x \\
 & \qquad (\mathit{from}_{\mathsf{Tree}}\ t_2)) \\
\mathit{to}_{\mathsf{Tree}} :: \ldots &
\end{array}
$$

- The only thing we have to change in the definition of *children* to use the new view is to add its name to the type signature:

$$\mathit{children}\langle \mathsf{a} :: \star\ \mathbf{viewed}\ \mathsf{Fix} \rangle :: (\forall \mathsf{b}\,.\,\mathit{col}\langle \mathsf{a} \mid \mathsf{b} \rangle) \Rightarrow \mathsf{a} \rightarrow [\mathsf{a}]$$

- The structure of a data type is no longer perceived as a sum of products, but as the fixed point of a functor.
- For example, $\mathit{children}\langle[\mathsf{Int}]\rangle\ [1,2,3]$ yields $[[2,3]]$.

Remarks about the fixed point view

The fixed point view faithfully encodes the PolyP approach. And hence it does not support

- mutually recursive data types,
- data types with higher kinded arguments (GRose),
- nested data types.

Other views

Other views that can be supported in our framework are:

- A balanced sums view enables a more efficient encoding function.
- List-like sums and products views for functions that need a more regular structure type.
- A *BoilerPlate* view with which we can mimic the generic programming approach of Peyton Jones and Lämmel.

What is a generic view?

A generic view on data types consists of the following components:

- A set of *view types* $\mathcal{V}$ for representing data types.
  - In the standard view we have Unit, Sum and Prod; while in the fixed point we have Fix.
- A partial mapping from type declarations T to structure types u and a list of supporting Haskell type declarations $\{D_i\}^{i \in 1..n}$.
  - In the standard view the structure type is a sum of products with no supporting declarations; in the fixed point view the structure type applies the Fix data type to a functor declared in D_i.
- For each data type T in the domain of the view mapping, a pair of conversion expressions to_T and $from_T$ between values and structure values.

Properties of a generic view

We say a generic view $\mathcal{V}$ is valid if the following properties hold:

- Kind preservation: the structure type has the same kind as the original type.
- The generated conversion expressions to_T and $from_T$ must be well typed,
- and their composition must give the identity function $to_D\ (from_D\ v) \equiv v$.

From the first two properties it follows that the specializations generated for generic functions are well typed. This guarantees, for example, that the specialization $col\langle \mathsf{Tree} \rangle$ has type:

$$col_{\mathsf{Tree}} :: \forall \mathsf{a}\ \mathsf{c}\,.\,(a \to [\mathsf{c}]) \to \mathsf{Tree}\ \mathsf{a} \to [\mathsf{c}]$$

Fixed point view mapping

$$\boxed{\mathcal{F}[\![D_0]\!]^{\mathrm{str}} == u; \{D_i\}^i_,}$$

$$\frac{D \equiv \mathbf{data}\ T = \{\Lambda a_i :: \star\ .\ \}^i\ \{C_j\ \{t_{j,k}\}^k\}^j_|}{\mathbf{type}\ \mathcal{F}[\![D]\!]^{\mathrm{str}} == \{\Lambda a_i :: \star\ .\ \}^i\ \mathsf{Fix}\ (\mathsf{ptr}(T)\ \{a_i\}^i); [\![D]\!]^{\mathrm{ptr}}}$$

$$\boxed{[\![D_1]\!]^{\mathrm{ptr}} == D_2}$$

$$\frac{\{a_{\ell+1}\ \not\equiv a_i\}^i \qquad \{\{t'_{j,k}\ \equiv [a_{\ell+1}\ /\ T\ \{a_i\}^i]\ t_{j,k}\}^k\}^j \qquad D \equiv \mathbf{data}\ \mathsf{ptr}(T) = \{\Lambda a_i :: \star\ .\ \}^i\ \Lambda a_{\ell+1} :: \star\ .\ \{\mathsf{ptr}(C_j)\ \{t'_{j,k}\}^k\}^j_|}{[\![\mathbf{data}\ T = \{\Lambda a_i :: \star\ .\ \}^i\ \{C_j\ \{t_{j,k}\}^k\}^j_|]\!]^{\mathrm{ptr}} \equiv D}$$

Conclusions and future work

- Generic views allow us to use different generic programming styles in a single framework.
- Some generic functions become simpler or more efficient.
- Adding a new view requires making a few localized changes to the compiler
- We intend to investigate generic views for type-indexed data types.
- Generic views are available in a branch of the Generic Haskell repository:

```
svn co https://svn.cs.uu.nl:12443/repos\
/Generic-Haskell/branches/GenericViews
```