Generic aspects of a structure editor

Johan Jeuring
Utrecht University
johanj@cs.uu.nl

February 2004

Joint work with Rui Guerra and Martijn Schrage
The title listed in the program is:

**Implementing a generic editor**

But in hindsight that was a bit too ambitious…
Introduction

→ In this talk I want to give an introduction to generic programming in Generic Haskell.
→ Most of the examples I use describe the generic aspects of a structure editor.
→ I will also give a brief overview of the Generic Haskell project.
Generic programming in Haskell
Generic Programming

 ➤ A generic program is written once, and works on values of many data types.
 ➤ Example: a generic equality program says that two values are equal provided the top nodes are equal, and that the top nodes have equally many children, which are pairwise equal.
 ➤ Other examples: map, fold, parse, pretty-print, serialise, and zip.
 ➤ A generic function is not necessarily a parametric polymorphic or an adhoc polymorphic function:

\[
\text{equal} :: \forall a. a \rightarrow a \rightarrow \text{Bool} \\
\text{equal} :: Eq a \Rightarrow a \rightarrow a \rightarrow \text{Bool} \\
\text{equal}\langle t :: \star \rangle :: t \rightarrow t \rightarrow \text{Bool}.
\]
Equality

→ The equality function on lists:

\[
\text{data } \text{IList} = \text{INil} \mid \text{ICons } \text{Int } \text{IList}
\]

\[
\text{eqILList} :: \text{IList } \rightarrow \text{IList } \rightarrow \text{Bool}
\]

\[
\text{eqILList } \text{INil } \text{INil } = \text{True}
\]

\[
\text{eqILList } (\text{ICons } x \text{ xs}) (\text{ICons } y \text{ ys}) = \text{eqInt } x \text{ y } \land \text{eqILList } \text{xs } \text{ys}
\]

\[
\text{eqILList } _{} _{} = \text{False}.
\]

→ The equality function on trees:

\[
\text{data } \text{ITree} = \text{ILeaf } \text{Int} \mid \text{IBin } \text{ITree } \text{ITree}
\]

\[
\text{eqITree} :: \text{ITree } \rightarrow \text{ITree } \rightarrow \text{Bool}
\]

\[
\text{eqITree } (\text{ILeaf } i) (\text{ILeaf } j) = \text{eqInt } i \text{ j}
\]

\[
\text{eqITree } (\text{IBin } l \text{ r}) (\text{IBin } v \text{ w}) = \text{eqITree } l \text{ v } \land \text{eqITree } r \text{ w}
\]

\[
\text{eqITree } _{} _{} = \text{False}.
\]
The structure of types

→ Generic Haskell adds to Haskell the ability to define a function by induction on the structure of types.

\[
\begin{align*}
\text{data } \text{IList} & = \text{INil} \mid \text{ICons} \text{ Int IList} \\
\text{type } \text{IList}' & = \text{Unit} + \text{Int} \times \text{IList} \\
\text{data } \text{ITree} & = \text{ILeaf} \text{ Int} \mid \text{IBin} \text{ ITree ITree} \\
\text{type } \text{ITree}' & = \text{Int} + \text{ITree} \times \text{ITree}
\end{align*}
\]

→ A data type is a sum of products of base types like Int or Char, or references to data types.

\[
\begin{align*}
\text{data } \text{Unit} & = \text{Unit} \\
\text{data } a + b & = \text{Inl} \ a \mid \text{Inr} \ b \\
\text{data } a \times b & = (a, b)
\end{align*}
\]
Generic equality

Here is the definition of function \textit{equal} in Generic Haskell.

\[
\begin{align*}
\text{equal}\langle \text{Unit}\rangle \: \text{Unit} \: \text{Unit} &= \text{True} \\
\text{equal}\langle \text{Int}\rangle \: i \: j &= \text{eqInt} \: i \: j \\
\text{equal}\langle a + b\rangle \: (\text{Inl} \: a_1) \: (\text{Inl} \: a_2) &= \text{equal}\langle a\rangle \: a_1 \: a_2 \\
\text{equal}\langle a + b\rangle \: (\text{Inr} \: b_1) \: (\text{Inr} \: b_2) &= \text{equal}\langle b\rangle \: b_1 \: b_2 \\
\text{equal}\langle a + b\rangle \: _\: _ &= \text{False} \\
\text{equal}\langle a \times b\rangle \: (a_1, b_1) \: (a_2, b_2) &= \text{equal}\langle a\rangle \: a_1 \: a_2 \land \text{equal}\langle b\rangle \: b_1 \: b_2
\end{align*}
\]

The type arguments are \langle specially marked\rangle.
The type of generic equality

- Function `equal` has the following type if it is indexed by types of kind `⋆`.

```
\text{equal}(t :: \star) :: t \rightarrow t \rightarrow \text{Bool}
```

- The type of `equal` is kind-indexed:

```
\text{equal}(t :: \kappa) :: \text{Equal} \langle \kappa \rangle t

\text{type Equal} \langle \star \rangle t = t \rightarrow t \rightarrow \text{Bool}

\text{type Equal} \langle \kappa \rightarrow \nu \rangle t = \forall u. \text{Equal} \langle \kappa \rangle u \rightarrow \text{Equal} \langle \nu \rangle (t u)
```

- For example:

```
\text{data List } a = \text{Nil} \mid \text{Cons } a \ (\text{List } a)

\text{equal} \langle \text{List} \rangle :: (a \rightarrow a \rightarrow \text{Bool}) \rightarrow \text{List } a \rightarrow \text{List } a \rightarrow \text{Bool}
```
Generic Haskell is an extension of Haskell that supports the construction of generic programs.

Generic Haskell adds to Haskell the notion of structural polymorphism, the ability to define a function (or a type) by induction on the structure of types.

Generic Haskell is implemented as a preprocessor that translates (by specialisation) generic functions into Haskell.

Most of the ideas go back to Ralf Hinze’s several papers about generic programming in Haskell.

You can obtain the compiler from www.generic-haskell.org.
Editors for structured documents
Classes of editors

➡️ Flat Editors.
Editors that are not aware of the structure (or type) of a document. Examples: Notepad, emacs.

➡️ Structure editors
- Syntax-directed.
The editing model is directly based on the structure. Example: the Cornell Program Synthesizer.
- Syntax-recognizing.
A document is edited in text mode and the editor infers the structure using an analysis. Example: Pan.
- Hybrid.
Structure editing enhanced with text editing capabilities.

➡️ With the advent of XML, structure editors have reappeared as XML editors. There are many XML editors, Google gives almost a million pages about XML editors. XML editors are generally pure syntax-directed editors.
Generic editing functionality

We expect several things from an editor:

→ navigation functionality over the document.
→ insert, cut, copy, paste, delete.
→ undo.

In order to implement this functionality we need (amongst others):

→ Placeholders and/or default values in documents.
→ A path to the selected subtree.
→ A clipboard for storing a subtree.
→ a history of states of the editor.
A type of editors

The type of an editor is given by:

data Editor t = Editor
{ doc :: t
, path :: Path
, clipboard :: ClipBoard
, history :: History
}. 
Default documents

When we start constructing a new value of a type, we either want a placeholder for the document, or a default document, which we can subsequently edit.

 chù The default document of type Int is 0.
 chù The default document of type Tree Int is Leaf 0, with

```haskell
data Tree a = Leaf a
               | Bin (Tree a) (Tree a).
```
A generic function for defaults

→ Here is a generic function for constructing a default value of a type.

\[
\text{type } \text{Default}\langle\star\rangle t = t
\]

\[
\text{type } \text{Default}\langle\kappa \to \nu\rangle t = \forall u.\text{Default}\langle\kappa\rangle u \to \text{Default}\langle\nu\rangle (t \ u)
\]

\[
\text{default}\langle t :: \kappa\rangle :: \text{Default}\langle\kappa\rangle t
\]

\[
\text{default}\langle\text{Unit}\rangle = \text{Unit}
\]

\[
\text{default}\langle\text{Int}\rangle = 0
\]

\[
\text{default}\langle\text{String}\rangle = ""
\]

\[
\text{default}\langle a + b\rangle = \text{Inl default}\langle a\rangle
\]

\[
\text{default}\langle a \times b\rangle = (\text{default}\langle a\rangle, \text{default}\langle b\rangle)
\]
An alternative for defaults is placeholders, which are used to represent empty documents. Ideally:

```haskell
data Tree a = Leaf a
  | Bin (Tree a) (Tree a)

data HoleTree a = Hole
  | HLeaf a
  | HBin (HoleTree a) (HoleTree a).
```
Type-indexed data types

- A type-indexed data type is a data type that is constructed in a generic way from an argument data type.
- Type-indexed data types are used in many applications, such as generic search trees, the zipper, and a generic editor.
The Hole data type

- A type-indexed data type defines a type on all structure types. This definition is translated to a set of newtypes, with the user supplied constructor names.

  \[
  \begin{align*}
  \textbf{type} \ Hole\langle\text{Unit}\rangle &= HUnit \ \text{Unit} \\
  \textbf{type} \ Hole\langle\text{Int}\rangle &= HInt \ \text{Int} \\
  \textbf{type} \ Hole\langle\text{String}\rangle &= HString \ \text{String} \\
  \textbf{type} \ Hole\langle a + b \rangle &= HSum \ (\text{Sum Unit (Sum Hole\langle a \rangle Hole\langle b \rangle)}) \\
  \textbf{type} \ Hole\langle a \times b \rangle &= HProd \ (\text{Prod Hole\langle a \rangle Hole\langle b \rangle})
  \end{align*}
  \]

- Type-indexed data types are only used by type-indexed functions.
Embedding $t$ in $\text{Hole}(t)$

Function $\text{toHole}$ embeds a value of type $t$ into $\text{Hole}(t)$.

\[
\begin{align*}
\text{type } \text{ToHole}\langle\star\rangle t &= t \to \text{Hole}(t) \\
\text{type } \text{ToHole}\langle\kappa \to \nu\rangle t &= \forall u. \text{ToHole}\langle\kappa\rangle u \to \text{ToHole}\langle\nu\rangle (t u) \\
toHole\langle t :: \kappa\rangle &:: \text{ToHole}\langle\kappa\rangle t \\
toHole\langle\text{Unit}\rangle u &= \text{HUnit} u \\
toHole\langle\text{Int}\rangle i &= \text{HInt} i \\
toHole\langle\text{String}\rangle s &= \text{HString} s \\
toHole\langle a + b \rangle (\text{Inl } a) &= \text{HSum} (\text{Inr} (\text{Inl} (\text{toHole}\langle a \rangle a))) \\
toHole\langle a + b \rangle (\text{Inr } b) &= \text{HSum} (\text{Inr} (\text{Inr} (\text{toHole}\langle b \rangle b))) \\
toHole\langle a \times b \rangle (a, b) &= \text{HProd} (\text{toHole}\langle a \rangle a, \text{toHole}\langle b \rangle b)
\end{align*}
\]
Too many holes

The type-indexed data type for holes introduces too many holes. For example, for the data type

\[
\textbf{data} \ \text{Expr} = \ Con \ \text{Int} \\
| \ Add \ Expr \ Expr \\
| \ Mul \ Expr \ Expr
\]

we get

\[
\textbf{data} \ \text{HoleExpr} = \ Hole1 \\
| \ HCon \ \text{Int} \\
| \ \text{Hole2} \\
| \ HAdd \ \text{HoleExpr} \ \text{HoleExpr} \\
| \ HMul \ \text{HoleExpr} \ \text{HoleExpr}
\]
A new definition of Hole

To avoid multiple holes in a data type with at least three constructors, the type-indexed data type Hole depends on the identity type-indexed data type IdHole.

dependency Hole ← Hole IdHole

\[
\begin{align*}
\text{type } & \text{Hole} \langle a + b \rangle = HSum \ (\text{Sum Unit (Sum IdHole} \langle a \rangle \ \text{IdHole} \langle b \rangle )) \\
\text{type } & \text{Hole} \langle a \times b \rangle = HProd \ (\text{Prod Hole} \langle a \rangle \ \text{Hole} \langle b \rangle )) \\
dependency \ & \text{IdHole} ← \text{IdHole Hole} \\
\text{type } & \text{IdHole} \langle a + b \rangle = IHSum \ (\text{Sum IdHole} \langle a \rangle \ \text{IdHole} \langle b \rangle )) \\
\text{type } & \text{IdHole} \langle a \times b \rangle = IHProd \ (\text{Prod Hole} \langle a \rangle \ \text{Hole} \langle b \rangle ))
\end{align*}
\]
The path of a document determines the *focus* in the document. It points to a subdocument.

For example, given the value 
\( Bin \left( Bin \left( Leaf\ 3 \right) \left( Leaf\ 6 \right) \right) \left( Leaf\ 8 \right) \), and the path \([L, R]\), the subtree \( Leaf\ 6 \) is in focus.

To determine the subdocument in focus, we have to record which child of a constructor is selected. Since children of constructors are represented by binary products, we use the following type for paths:

```haskell
data BinDirect = L | R
type Path = [BinDirect].
```
There are four basic navigation directions: down, up, right, left:

\[
\text{data NavDirect} = \text{NUp} \mid \text{NDown} \mid \text{NLeft} \mid \text{NRight}.
\]

A call to the \textit{navigate} function updates the path of a document given a navigation direction:

\[
\text{navigate} \langle t :: \kappa \rangle :: \text{Navigate}\langle \kappa \rangle t
\]

\textbf{type \textit{Navigate}}\langle \star \rangle t = \text{Editor Hole}\langle t \rangle \to \text{NavDirect} \to \text{Maybe Path}

\textbf{type \textit{Navigate}}\langle \kappa \to \nu \rangle t = \forall u.\text{Navigate}\langle \kappa \rangle u \to \text{Navigate}\langle \nu \rangle (t u).
The zipper

- At each navigation step, the complete path is used to change the state of the editor. This is a bit inefficient. An alternative approach would be to use the zipper.
- The zipper is a data structure that is used to represent a tree together with a subtree that is the focus of attention, where that focus may move left, up, down, or right in the tree.
- The zipper structure has been described by Huet, who uses it in a structure editor. All operations of the zipper are cheap.
The zipper on trees

For each type, we have to construct its zipper type. For example:

```haskell
data ITree = ILeaf Int | IBin ITree ITree

type Loc_ITree = (ITree, Context_ITree)

data Context_ITree = Top
                 | LBin Context_ITree ITree
                 | RBin ITree Context_ITree

left_ITree, ... :: Loc_ITree → Loc_ITree.
```

So to implement the zipper we need a type-indexed data type that returns the zipper type of the argument type.
Navigating on trees

Using the location type, we can efficiently navigate through trees.

\[
\text{down}_{\text{ITree}} :: \text{Loc}_{\text{ITree}} \rightarrow \text{Loc}_{\text{ITree}}
\]

\[
\text{down}_{\text{ITree}} (\text{ILeaf } a, c) = (\text{ILeaf } a, c)
\]

\[
\text{down}_{\text{ITree}} (\text{IBin } l r, c) = (l, \text{LBin } c r)
\]

\[
\text{right}_{\text{ITree}} :: \text{Loc}_{\text{ITree}} \rightarrow \text{Loc}_{\text{ITree}}
\]

\[
\text{right}_{\text{ITree}} (\text{tl}, \text{LBin } c tr) = (tr, \text{RBin } tl c)
\]

\[
\text{right}_{\text{ITree}} l = l
\]

Note that function \textit{down} is defined by pattern matching on the tree in focus, and function \textit{right} by pattern matching on the context.

Functions \textit{left}_{\text{ITree}} and \textit{up}_{\text{ITree}} are defined similarly.
Navigating on data types

The navigation functions from the zipper may only move to recursive components. For example, if we select the left subtree in a `NLBin` constructor from

\[
\text{data NLTree } a = NLLeaf \text{ Char} \\
| \text{NLBin } (\text{NLTree } a) \ a \ (\text{NLTree } a)
\]

and we try to move right, we move to the next NLTree, and not to the value of type \( a \).

Recursive positions play an important role in the zipper. To access recursive positions, we define data types as fixed-points of functors.

Disadvantages of using the zipper:

- the zipper only works on data types that can be defined as a fixed-point of a functor.
- it is impossible to have multiple foci.
Function *delete* substitutes a placeholder (a hole) for the subdocument in focus.

\[
delete\langle t :: \kappa \rangle :: \text{Delete}\langle\kappa\rangle \cdot t
\]

**Type**

\[
\text{type} \ \text{Delete}\langle\ast\rangle :: t = \text{Editor Hole}\langle t\rangle \rightarrow \text{Editor Hole}\langle t\rangle
\]

\[
\text{type} \ \text{Delete}\langle\kappa \rightarrow \nu\rangle :: t = \forall u.\text{Delete}\langle\kappa\rangle \cdot u \rightarrow \text{Delete}\langle\nu\rangle \cdot (t \cdot u)
\]

\[
delete\langle a \times b \rangle \cdot (\text{Editor} \ (a, b) \cdot \text{path} \ cb \ h) =
\begin{align*}
\text{if} & \ \text{null} \ \text{path} \\
\text{then} & \\
& \text{Editor} \ (\text{HSum} \ (\text{Inl} \ \text{Unit})) \cdot \text{path} \ cb \ h \\
\text{else} & \\
& < \text{follow} \ \text{path} \ \text{until} \ \text{empty} >
\end{align*}
\]
The ClipBoard

→ A clipboard is used to store subtrees by means of functions *copy* and *cut*, and to paste subtrees by means of function *paste*.

→ Since a clipboard may contain values of any data type, we use the type Dynamic for a value in the clipboard.

```plaintext
type ClipBoard = Dynamic
```
Copying to the clipboard

Function \textit{copy} copies a value to the clipboard.

\[
\text{copy}\langle t :: \kappa \rangle :: \text{Copy}\langle \kappa \rangle t \\
\text{type } \text{Copy}\langle \star \rangle t = \text{Editor Hole}\langle t \rangle \rightarrow \text{Clipboard}
\]

\[
\text{type } \text{Copy}\langle \kappa \rightarrow \nu \rangle t = \forall u. \text{Copy}\langle \kappa \rangle (u) \rightarrow \text{Copy}\langle \nu \rangle (t \ u)
\]

\[
\text{copy}\langle \text{Int} \rangle (\text{Editor } (H\text{Int } i) \ \text{path } cb \ h) = \text{toDyn } i
\]

Function \textit{cut} is almost the same, except for the fact that it also replaces the current focus by a hole. Here is part of its type:

\[
\text{type } \text{Cut}\langle \star \rangle t = \text{Editor Hole}\langle t \rangle \rightarrow \text{Editor Hole}\langle t \rangle.
\]

Function \textit{cut} can be implemented as the composition of \textit{delete} after \textit{copy}.
Pasting from the clipboard

\[
paste(t :: \kappa) :: Paste\langle \kappa \rangle t
\]

\[
\text{type Paste}\langle \star \rangle t = \text{Editor Hole}\langle t \rangle \rightarrow \text{Editor Hole}\langle t \rangle
\]

\[
\text{type Paste}\langle \kappa \rightarrow \nu \rangle t = \forall u. Paste\langle \kappa \rangle u \rightarrow Paste\langle \nu \rangle (t u)
\]

\[
paste(\text{Int}) e@ (\text{Editor} (HInt s) \text{ path cb h}) =
\]

\[
\text{case cb of}
\]

\[
("\text{Int"}, v) \rightarrow \text{Editor} (HInt v) \text{ path cb h}
\]

\[
(\_ , \_ ) \rightarrow e
\]
The history

There are at least two possibilities for encoding the history:

- Let the history be an editor again, representing the previous state.
- Define inverses of the editing operations, and for each editing operation store its inverse in the history.
Presentation-oriented editing

In a hybrid editor we want to be able to directly navigate and edit on the presentation of a document. Furthermore, we want to be able to store invalid documents. In order to do so, we have to add a new constructor to each data type:

```
data Expr = Con Int
  | Add Expr Expr
  | Mul Expr Expr
  | ParseError String
```

so that for example the string $2 + (4 \times 15) + \ast 2$ is represented by

```
Add (Con 2) (Add (Mul (Con 4)
  (Con 15))
  (ParseError "\ast 2")
).
```
Proxima

Proxima is a presentation-oriented generic structure editor, with the following characteristics:

- Generic.
- Computations over the document.
- Advanced graphical presentation formalism.
- Editing on both the presentation and the document, and even on the derived values.
- Modeless editing.
- Support for extra state.

More information can be found in Martijn Schrage’s forthcoming PhD thesis (May/June, Utrecht University).
The generic features of Proxima

Proxima doesn’t use Generic Haskell, but instead a special-purpose generator that generates placeholder data types etc.

The work discussed in this talk is an attempt to (maybe partially) replace Proxima’s generator by generic programs.

There are still some problems with the approach sketched in this talk:

- The fact that data types are viewed as binary sums of binary products is not always convenient when writing generic editing programs.
- If we want a precise specification of the Hole type-indexed data type, we get a dependency on IdHole. This dependency is reflected in all functions defined on values with placeholders. The same holds for the type-indexed data type for representing parse errors.
Overview of the Generic Haskell project
The Generic Haskell project

The Generic Haskell project started in 2000, and will finish in 2004. Currently, the people involved are: Johan Jeuring, Ralf Hinze, Andres Löh, and Frank Atanassow, together with some students.

The project consists of three subprojects:

- Programming language and compiler.
- Theory.
- Applications of generic programming.
Generic Haskell: programming language

In the Generic Haskell programming language we can write:

- Generic (type-indexed) functions with kind-indexed types.

- Type-indexed data types.

- Dependencies between generic functions.

- Cases for specific types and constructors.
Generic Haskell: compiler


- The generated code is still rather inefficient. Efficient code can be obtained via partial evaluation. An MSc student has produced an optimiser for Generic Haskell, which seems to remove all overhead caused by the Generic Haskell translation. The Clean compiler (a Haskell-like functional programming language with support for generic programming), generates very efficient code for generic functions.
Generic Haskell: theory

- The theory of (dependency-style) Generic Haskell is described in Ralf Hinze’s habilitationsschrift, and in Andres Löh’s forthcoming PhD thesis (August/September 2004).
Applications of generic programming

- A generic dictionary, compressor, and zipper.

- XML data binding.
  Frank Atanassow, Dave Clarke, and Johan Jeuring. Scripting

- Inferring type isomorphisms generically.
  Frank Atanassow, and Johan Jeuring. Submitted for
An XML Haskell data binding

- We have developed an XML Haskell data binding: a tool that translates an XML Schema to a Haskell data type, preserving the typing information in the Schema, together with generic functions for pretty-printing and parsing XML documents.

- The data binding creates rather unwieldy data types. We can generically infer an isomorphism between two data types, provided they are isomorphic. So supplying the desired, isomorphic, data type for a binding is sufficient.
Conclusions

- Introduction to generic programming in Generic Haskell.
- There are quite a number of generic aspects in a structure editor. New aspects of the editor (compared with the zipper): a type-indexed data type for holes, a clipboard, and multiple foci are possible after extending the Path data type.
- It is possible to implement the generic aspects of a structure editor in Generic Haskell. This implementation is sometimes elegant, sometimes rather involved. To obtain better code, Generic Haskell needs a more flexible approach to representing data types.
- Within the Generic Haskell project we also study XML-Haskell data bindings, type isomorphisms, and all kinds of language related topics.