Generic Dictionaries

Johan Jeuring

August 26, 2002

Why generic programming?

Why generic programming?

Because programs become easier to write!

Why generic programming?

Because programs become easier to write!

Why is it easier to write programs?

- Programs that you could only write in an untyped style have types now.
- Some programs are for free.
- Some programs are simple adjustments of library functions.

Introduction

Ralf has introduced the concept of a generic (or polytypic, type-indexed) function: a function that can be instantiated on all Haskell data types to obtain data specific functionality.

Over the last few years we have worked on generic programming and **Generic H∀SKELL**,see

```
http://www.generic-haskell.org/
```

The development of **Generic H∀SKELL** has to a large extent been example driven. In my lectures I will talk about three larger examples:

- Generic dictionaries
- XComprez, a compressor for XML documents
- The Zipper

Applications of generic programming

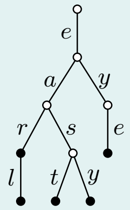
Applications of generic programming are found in several fields:

- Haskell's deriving construct: equality, read/show, ...
- Compiler tools: debuggers, garbage collectors, tracers.
- XML tools: editors, compressors, database tools.
- Typed term processing: rewriting, unification, pattern matching.
- Tree traversals.
- . . .

Almost all of these fields have in common that structure drives the application. The programs are 'syntax-directed'.

Introduction Generic Dictionaries

A trie is a search tree scheme that employs the structure of a search key to organize information. For example, the set of strings $\{ear, earl, east, easy, eye\}$ can be represented by the following tree:



Overview

In this talk I will

- Define a module for generic dictionaries.
- Introduce type-indexed data types and kind-indexed kinds.
- Introduce default cases and constructor cases.
- Introduce dependencies and generic abstractions.

Dictionaries on strings

The data type of strings and dictionaries on strings can be defined as follows:

```
data String          =  Nil | Cons Char String

data FMap_String v   =
  Trie_String (Maybe v) (DictChar (FMap_String v))
```

where DictChar is a dictionary on characters. Here is the example:

```
eDict    ::  FMap_String ()
eDict    =   Trie_String Nothing cDict1

cDict1   ::  DictChar (FMap_String ())
cDict1   =   [('e',eDict1)]

eDict1   =   Trie_String Nothing cDict2
cDict2   =   [('a',eDict2),('y',eDict3)]
```

Looking up strings

Here is how you look up a string value.

```
lookup_String  ::  String -> FMap_String v -> Maybe v
lookup_String Nil          (Trie_String tn tc)  =  tn
lookup_String (Cons c s) (Trie_String tn tc)  =
   lookup_Char c tc   >>=  \t -> lookup_String s t

lookup_Char  ::  Char -> DictChar v -> Maybe v
```

Looking up characters

We implement a dictionary for characters as an ordered association list:

```
type DictChar v = [(Char,v)]

lookup_Char  ::  Char -> DictChar v -> Maybe v
lookup_Char c []                          =  Nothing
lookup_Char c ((c',v):xs) | c <  c'   =  Nothing
                          | c == c'   =  Just v
                          | c >  c'   =  lookup_Char c xs
```

Dictionaries on trees

A dictionary on trees is a value of the following data type.

```
data Tree a            =  Leaf | Node (Tree a) a (Tree a)

data FMap_Tree fma v   =
  Trie_Tree (Maybe v) (FMap_Tree fma (fma (FMap_Tree fma v)))
```

The `lookup` function on tree dictionaries is defined as follows.

```
lookup_Tree  ::  (forall v.a -> fma v -> Maybe v) ->
                 Tree a -> FMap_Tree fma w -> Maybe w
lookup_Tree lua Leaf         (Trie_Tree tl tn)  =  tl
lookup_Tree lua (Node l m r) (Trie_Tree tl tn)  =
  lookup_Tree lua l tn  >>=  \tm ->
  lua m tm              >>=  \tr ->
  lookup_Tree lua r tr
```

A module for generic dictionaries

We want to have a function `FMap` that takes a data type and returns the dictionary type for that data type, together with functions for looking up in a dictionary, and manipulating a dictionary.

Furthermore, we want to have a number of functions on the generated dictionary data type

```
type FMap{| t |} ...

lookup{| t |}  ::  forall v . t -> FMap{| t |} v -> Maybe v
empty {| t |}  ::  FMap{| t |} v
single{| t |}  ::  (t,v) -> FMap{| t |} v
insert{| t |}  ::  (v -> v -> v) -> (t,v) ->
                   FMap{| t |} v -> FMap{| t |} v
delete{| t |}  ::  t -> FMap{| t |} v -> FMap{| t |} v
merge {| t |}  ::  (v -> v -> v) ->
                   FMap{|t|} v -> FMap{|t|} v -> FMap{|t|} v
...
```

A type-indexed trie type

Tries are based on the following isomorphisms, also known as the laws of exponentials.

$$\begin{aligned} 1 \rightarrow_{\text{fin}} V &\cong V \\ (T_1 + T_2) \rightarrow_{\text{fin}} V &\cong (T_1 \rightarrow_{\text{fin}} V) \times (T_2 \rightarrow_{\text{fin}} V) \\ (T_1 \times T_2) \rightarrow_{\text{fin}} V &\cong T_1 \rightarrow_{\text{fin}} (T_2 \rightarrow_{\text{fin}} V) \end{aligned}$$

In **Generic H∀SKELL**, a type-indexed data type for dictionaries is defined as follows.

```
type FMap{| Unit |}          v   =   FMUnit (Maybe v)
type FMap{| Char |}          v   =   FMChar (DictChar v)
type FMap{| :+:  |} fma fmb v   =   FMEither (fma v,fmb v)
type FMap{| :*:  |} fma fmb v   =   FMProd (fma (fmb v))
```

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 given at the beginning of this lecture.

Besides finishing the example, I will also discuss the ‘type’ of type-indexed data types.

Type-indexed data types are just as important as type-indexed functions!

Kind-indexed kinds

What is the type of a type-indexed data type? The type of a data type is a kind. But a type-indexed data type depends on the kind of the input type. So a type-indexed data type has a kind-indexed kind!

```
type FMap{| t :: k |}  ::  FMAP{[ k ]}

FMAP{[  *   ]}  =  * -> *
FMAP{[ k->l ]}  =  FMAP{[ k ]} -> FMAP{[ l ]}
```

Function `lookup`

The generic function `lookup` is defined as follows:

```
lookup{| t :: k |}  ::  Lookup{[ k ]} t

type Lookup{[  *   ]} t  =
  forall v.t -> FMap{| t |} v -> Maybe v
type Lookup{[ k->l ]} t  =
  forall a.Lookup{[ k ]} a -> Lookup{[ l ]} (t a)
```

Function `lookup`

The generic function `lookup` is defined as follows:

```
lookup{| t :: k |}  ::  Lookup{[ k ]} t

type Lookup{[  *   ]} t  =
  forall v.t -> FMap{| t |} v -> Maybe v
type Lookup{[ k->l ]} t  =
  forall a.Lookup{[ k ]} a -> Lookup{[ l ]} (t a)
```

```
lookup{| Unit |}          Unit    (FMUnit v)           =  v
lookup{| Char |}          c       (FMChar t)           =
  lookup_Char c t
lookup{| :+:  |} lua lub (Inl a) (FMEither (t1,t2))  =
  lua a t1
lookup{| :+:  |} lua lub (Inr b) (FMEither (t1,t2))  =
  lub b t2
lookup{| :*:  |} lua lub (a:*:b) (FMProd t)           =
  lua a t  >>=  \t' -> lub b t'
```

Default cases

Suppose I have a more efficient lookup function for characters `lookup_Char_efficient`. Then I can reuse `lookup` by means of a *default case*:

```
lookup'{| t :: k |}     :: Lookup{[ k ]} t

lookup'{| Char |} c t   =  lookup_Char_efficient c t
lookup'{|   a   |}      =  lookup{| a |}
```

Constructor cases

Suppose I want to use the efficient lookup function only for one occurrence of the type Char in my data type. Then I can use a *constructor case*:

```
data Example = C1 Char | C2 Char | ....

lookup'{| t :: k |}                :: Lookup{[ k ]} t

lookup'{| case C1 |} (C1 c) t  =  lookup_Char_efficient c t
lookup'{|   a      |}           =  lookup{| a |}
```

Function `empty`

The generic function `empty` is defined as follows:

```
empty{| t :: k |}  ::  Empty{[ k ]} t

type Empty{[  *   ]} t  =  forall v.FMap{| t |} v
type Empty{[ k->l ]} t  =
  forall a.Empty{[ k ]} a -> Empty{[ l ]} (t a)

empty{| Unit |}            =  FMUnit Nothing
empty{| Char |}            =  FMChar []
empty{| :+:  |} ea eb      =  FMEither (ea,eb)
empty{| :*:  |} ea eb      =  FMProd ea
```

Function single: dependencies

The generic function single depends on the function empty:

```
dependency single <- single empty

type Single{[ * ]}    t  =  forall v.(t,v) -> FMap{| t |} v
type Single{[ k->l ]} t  =
  forall a.Single{[k]} a -> Empty{[k]} a -> Single{[l]} (t a)

single{| t :: k |}                          :: Single{[ k ]} t
single{| Unit |}             (Unit,v)       =  FMUnit (Just v)
single{| Char |}             (c,v)          =  FMChar [(c,v)]
single{| :+: |} sA eA sB eB (Inl a,v)       =
  FMEither (sA (a,v),eB)
single{| :+: |} sA eA sB eB (Inr b,v)       =
  FMEither (eA,sB (b,v))
single{| :*: |} sA eA sB eB (a :*: b,v)     =
  FMProd (sA (a,sB (b,v)))
```

Function insert: generic abstractions

The generic function insert can be defined as a generic abstraction of single and merge:

```
insert{| t :: * |}         :: (v -> v -> v) -> (t,v) ->
                                FMap{| t |} v -> FMap{| t |} v
insert{| t |} c (x,v) d  =
  merge{| t |} c (single{| t |} (x,v)) d
```

The other functions on generic dictionaries are implemented in a similar fashion.

Exercise

The definition of function `insert` as a generic abstraction of `single` and `merge` limits its application to data types of kind $*$. Define `insert` as a type-indexed function with a kind-indexed type.

The code for the generic dictionaries library can be downloaded from

`http://www.generic-haskell.org/applications/`

The solution to the exercise is at the end of the file FMap.ghs.

But first try to solve one of the simpler exercises from the practice and theory part, or exercise 1 in the applications.

Conclusions

I have shown:

- How you can define a module for generic dictionaries.
- How you can define type-indexed data types in **Generic H∀SKELL**.
- How you can use default cases, constructor cases, dependencies, and generic abstractions in **Generic H∀SKELL**.

Next lectures

- XComprez
- The Zipper

Generic Programming for XML Tools

Johan Jeuring

August 27, 2002

XML Tools

Since W3C released XML, hundreds of XML tools have been developed.

- XML editors (XML Spy, XMetal)
- XML databases (XHive)
- XML converters, parsers, and validators
- XML version managament tools (XML Diff and Merge, IBM treediff)
- XML compressors
- XML encryption tools

Usage of DTDs in XML Tools

Some XML tools critically depend on, or would benefit from knowing, the DTD of a document. We call such a tool *DTD indexed*. Examples of DTD-indexed tools are:

- XML editors
- XML validators/parsers
- XML databases
- XML compressors
- XML version management tools

Generic Programming for XML Tools

Since DTD-indexed XML tools are generic programs, it would help to implement such tools as generic programs:

- *Development time*.
  - DTD processing by the compiler.
  - Library of basic generic programs.
- *Correctness*. Valid documents will be transformed to valid documents, possibly structured according to another DTD.
- *Efficiency*. The generic programming compiler may perform all kinds of optimisations on the code.

Overview

This talk:

- DtdToHaskell (HaXml)
- XComprez, an XML Compressor

Compressing XML files

XML is a very verbose standard, and compression might considerably reduce the size of XML files.

I know of four XML compressors:

- XMLZip
- XMill
- Millau
- XML-Xpress

XMill

Consider the following book example:

```
<book lang="English">
<title>   Dead famous   </title>
<author> Ben Elton     </author>
<date>    2001          </date>
<chapter>Introduction </chapter>
<chapter>Preliminaries</chapter>
</book>
```

XMill splits this document in contents and structure. The contents are collected in different containers for different elements, and the structure becomes:

```
book=#1, @lang=#2 title=#3 author=#4 date=#5 chapter=#6

#1 #2 C1 / #3 C2 / #4 C3 / #5 C4 / #6 C5 / #6 C5 / /
```

Using the DTD

Here is the book DTD:

```
<!ELEMENT book    (title,author,date,(chapter)*)>
<!ELEMENT title   (#PCDATA)>
<!ELEMENT author  (#PCDATA)>
<!ELEMENT date    (#PCDATA)>
<!ELEMENT chapter (#PCDATA)>
<!ATTLIST book lang (English | Dutch) #REQUIRED>
```

From the DTD you can see that you don't have to store the # 1, # 2, etc. information: the DTD enforces their presence. We only have to know how many chapters there are, and what the value of the `lang` attribute is. This implies you can compress better by taking the DTD into account.

XComprez

XComprez is a generic program for compressing values of data types. On a couple of example files it compresses almost twice as good as XMill. You can download XComprez from

```
http://www.generic-haskell.org/xmltools/XComprez
```

XComprez consists of four components:

- a component that translates a DTD to a data type
- a component that separates a value into structure and contents
- a component for compressing the structure
- a component for compressing the contents

HaXml

HaXml is a Haskell tool and a set of combinators for manipulating XML documents with Haskell. We use HaXml to generate a data type from a DTD, and to generate read and show functions from XML to values of the data type and vice versa. You can obtain HaXml from:

```
http://www.cs.york.ac.uk/fp/HaXml/
```

We will only use `DtdToHaskell.hs` from the HaXml library.

DtdToHaskell: DTD to data type

DtdToHaskell takes a DTD and generates a data type, and read and show functions from XML to values of the data type and vice versa. For example, for the book DTD you get:

```
data Book  =  Book Book_Attrs Title Author Date [Chapter]
  deriving (Eq,Show)
data Book_Attrs = Book_Attrs { bookLang :: Lang }
  deriving (Eq,Show)
data Lang = English  |  Dutch
  deriving (Eq,Show)
newtype Title    =  Title String    deriving (Eq,Show)
newtype Author   =  Author String   deriving (Eq,Show)
newtype Date     =  Date String     deriving (Eq,Show)
newtype Chapter  =  Chapter String  deriving (Eq,Show)
```

together with a function for reading (writing) an XML document into a value of this type.

DtdToHaskell: XML to value

The example XML document is read by the read function generated by DtdToHaskell into the following value:

```
exBook   :: Book
exBook   =  Book Book_Attrs{bookLang=English}
                (Title "  Dead famous   ")
                (Author " Ben Elton     ")
                (Date "   2001          ")
                [Chapter "Introduction "
                ,Chapter "Preliminaries"
                ]
```

Separating structure and contents

The contents of an XML document is obtained by extracting all `PCData` and `CData` from the document. In Generic Haskell, the contents of a value of a data type is obtained by extracting all strings from the value. For the above example value, we obtain the following result:

```
["  Dead famous  "
," Ben Elton   "
,"   2001         "
,"Introduction "
,"Preliminaries"
]
```

Generic extraction

The list of strings is obtained by applying

```
extract{| Book :: * |} exBook
```

This is an instance of the following generic function.

```
extract{| t :: * |}                   :: t -> [String]
extract{| Unit |}       Unit          =   []
extract{| String |}     s             =   [s]
extract{| :+: |} eA eB (Inl x)        =   eA x
extract{| :+: |} eA eB (Inr y)        =   eB y
extract{| :*: |} eA eB (x:*:y)        =   eA x ++ eB y
extract{| Con c |}    e (Con b)       =   e b
extract{| Label l |} e (Label b)      =   e b
```

The shape of a book

The structure of an XML document is obtained by removing all PCData and CData from the document. In Generic Haskell, the structure, or shape, of a value of a data type is obtained by replacing all strings by units (empty tuples). For example, the type we obtain from the data type Book is isomorphic to the following data type:

```
data SHAPEBook         =
  SHAPEBook SHAPEBook_Attrs SHAPETitle SHAPEAuthor
            SHAPEDate [SHAPEChapter]
data SHAPEBook_Attrs   =
  SHAPEBook_Attrs { bookLang :: SHAPELang }
data SHAPELang         =  SHAPEEnglish  |  SHAPEDutch
newtype SHAPETitle     =  SHAPETitle ()
newtype SHAPEAuthor    =  SHAPEAuthor ()
newtype SHAPEDate      =  SHAPEDate ()
newtype SHAPEChapter   =  SHAPEChapter ()
```

Another example of a type-indexed data type.

An example book shape

The shape of the example book is:

```
shapeBook  :: SHAPEBook
shapeBook  =
  SHAPEBook (SHAPEBOOK_Attrs { bookLang=SHAPEEnglish })
            (SHAPETitle ())
            (SHAPEAuthor ())
            (SHAPEDate ())
            [SHAPEChapter ()
            ,SHAPEChapter ()
            ]
```

The type-indexed data type SHAPE

```
type SHAPE{| Unit |}         =   SH1 Unit
type SHAPE{| String |}       =   SHString Unit
type SHAPE{| :+: |} sa sb    =   SHEither (Sum sa sb)
type SHAPE{| :*: |} sa sb    =   SHProd (Prod sa sb)
type SHAPE{| Con |} sa       =   SHCon (Con sa)
type SHAPE{| Label |} sa     =   SHLabel (Label sa)
```

Here is its kind-indexed type:

```
type SHAPE{| t :: k |} :: KSHAPE{[ k ]}

KSHAPE{[  *   ]}  =  *
KSHAPE{[ k->l ]}  =  KSHAPE{[ k ]} -> KSHAPE{[ l ]}
```

Function shape

Function shape returns the shape of a value, that is, it replaces all strings by units (empty tuples).

```
type Shape{[  *   ]} t  =  t -> SHAPE{| t |}
type Shape{[ k->l ]} t  =
  forall a. Shape{[ k ]} a -> Shape{[ l ]} (t a)

shape{| t :: k |}                  :: Shape{[ k ]} t
shape{| Unit |} u                  =  SH1 Unit
shape{| String |} s                =  SHString Unit
shape{| :+: |} sa sb (Inl a)       =  SHEither (Inl (sa a))
shape{| :+: |} sa sb (Inr b)       =  SHEither (Inr (sb b))
shape{| :*: |} sa sb (a :*: b)     =  SHProd (sa a :*: sb b)
shape{| Con c |} sa (Con b)        =  SHCon (Con (sa b))
shape{| Label l |} sa (Label b)    =  SHLabel (Label (sa b))
```

Function `insert`

Function `insert` takes a `SHAPE` value and a list of strings and inserts the strings at the right positions. It is the inverse of function `extract`.

```
type Insert{[  *   ]} t  =
  SHAPE{| t |} -> [String] -> (t,[String])
type Insert{[ k->l ]} t  =
  forall u . Insert{[ k ]} u -> Insert{[ l ]} (t u)

insert{| t :: k |}          :: Insert{[ k ]} t
insert{| Unit |} u ss        =  (Unit, ss)
insert{| String |} s ss      =  (head ss, tail ss)
insert{| :+: |} iA iB (SHEither (Inl a)) ss  =
  let {(ia,ss') = iA a ss} in (Inl ia, ss')
...
```

Encoding constructors

A constructor of a value of a data type is encoded as follows.

- Calculate the number n of constructors of the data type.
- Calculate the position of the constructor in the list of constructors of the data type.
- Replace the constructor by the bit representation of its position, using $log_2\ n$ bits.

For example, in a data type with 6 constructors, the third constructor is encoded by `010`. Note that we start counting with 0. Furthermore, note that a value of a data type with a single constructor is represented by 0 bits. So the values of all types except for `String` and `Lang` in the example are represented by 0 bits. The generic function `encode` has the following (slightly simplified) type:

```
encode{| t :: * |}  ::  SHAPE{| t |} -> [Bit]
```

The type of function encode

Here is the type of function encode. When computing the encoding, function encode also needs the list of constructors of the data type. For this purpose we use a dependency on constructors

```
dependency encode <- constructors encode

encode{| t :: k |}  ::  Encode{[ k ]} t

type Encode{[  *   ]} t      =
  Maybe [ConDescr] -> SHAPE{| t |} -> [Bit]
type Encode{[ k->l ]} t  =
  forall u . Constructors{[ k ]} u ->
             Encode{[ k ]} u      ->
             Encode{[ l ]} (t u)
```

The definition of function `encode` I

Encoding `Unit` and `String` is easy:

```
encode{| Unit |}    _ _  =  []
encode{| String |} _ _  =  []
```

A sum is encoded by calculating the constructors, and using them in the right component of the sum.

```
encode{| :+:  |} cA eA cB eB cs (SHEither (Inl a)) =
  let cs' = maybe (cA ++ cB) id cs
  in  eA (Just cs') a
```

The definition of function `encode` II

On constructors, `encode` calculates the bits which are returned in the bit list.

```
encode{| Con c |} cA eA cs (SHCon (Con a))  =
   let cs' = maybe [c] id cs
   in  intinrange2bits (length cs')
                       (fromJust (elemIndex c cs'))
       ++ eA Nothing a
```

Note the argument `Nothing` in the call of `eA`: the constructors used for encoding below a constructor might come from another data type.

Finally, products are encoded by calculating the encodings of the components.

```
encode{| :*:  |} cA eA cB eB _ (SHProd (a :*: b))  =
  eA Nothing a ++ eB Nothing b
```

Function `decode`

Function `decode` is the inverse of function `encode`: given a list of bits it produces a value of a shape of a data type. It has the following kind-indexed type.

```
dependency decode <- constructors decode

type Decode{[ * ]} t  =
  [Bit] -> Maybe [ConDescr] -> (Maybe SHAPE{| t |}, [Bit])
```

This function is used to calculate the shape of a value. Given the shape and a list of strings, we can use function `insert` to obtain a value of the data type corresponding to the original XML document. Showing this with the generated show function gives the original XML document.

Exercise: Huffman coding

If we know that some constructors occur much more often in a document than others, we can use this information to obtain better compression. For example, suppose we have the following list-like data type:

```
data FList a  =  Empty | Leaf a | Cons a (FList a)
```

The standard algorithm (XComprez) would use two bits for every constructor. However, any value of type `FList a` contains exactly one occurrence of either `Empty` or `Leaf`, and possibly many occurrences of `Cons`. So it is much better to use one bit for `Cons`, and two bits for the other constructors.

Implement a function that counts the number of occurrences of constructors in a value of a data type (see also the exercises in theory and practice). Use this function to implement Huffman coding.

Conclusions and Future work

Generic programming is useful for constructing XML tools: using HaXml/Generic Haskell allows for quick implementations of XML tools.

We are working on implementing several XML tools (XML editor,...)

Future work:

- Types get in the way in several places:
  - XSL rules contain match expressions that should be matched against any XML document.
  - Some of the DOM functions are hard to type in Generic Haskell.
- We want to be able to handle Schemata.

The Zipper

Johan Jeuring

August 29, 2002

Introduction

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.

In this talk I will:

- Introduce the zipper on an example data type.
- Define the zipper as a type-indexed data type.
- Define several navigation functions on the zipper.

The zipper is the most complicated generic program we have written.

The zipper on trees

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

```
data Tree   =  Leaf Char | Node Tree Tree

type Loc_Tree = (Tree, Context_Tree)

data Context_Tree   =  Top
                    |  LNode Context_Tree Tree
                    |  RNode Tree Context_Tree

left_Tree,... :: Loc_Tree -> Loc_Tree
```

So we want to have a function `Zipper` that takes a data type and returns a data type that represents the zipper type of the argument type.

Navigating on trees

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

```
down_Tree                   :: Loc_Tree -> Loc_Tree
down_Tree (Leaf a,c)        =  (Leaf a,c)
down_Tree (Node l r,c)      =  (l,LNode c r)

right_Tree                      :: Loc_Tree -> Loc_Tree
right_tree (tl,LNode c tr)      =  (tr,RNode tl c)
right_tree l                    =  l
```

Note that function `down` is defined by pattern matching on the tree in focus, and function `right` by pattern matching on the context.

Functions `left_Tree` and `up_Tree` 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 `NLNode` constructor from

```
data NLTree a  =  NLLeaf Char
               |  NLNode (NLTree a) a (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 obtain access to the recursive positions, we define data types as *fixed-points of functors*.

Data types as fixed points

The `Fix` data type is used to define data types as fixed points:

```
newtype Fix f = In { out :: f (Fix f) }
```

The type of trees can be defined as a fixed point as follows:

```
data Tree      =  Leaf Char | Node Tree Tree
data TreeF a   =  LeafF Char | NodeF a a

exTree  :: Tree
exTree  =  Node (Node (Leaf 'j') (Leaf 't')) (Leaf 'j')

exTreeF  :: Fix TreeF
exTreeF  =
  In (NodeF (In (NodeF (In (LeafF 'j')) (In (LeafF 't'))))
            (In (LeafF 'j')))
```

Converting between data types and fixed points

We can easily convert between data types an their representations as fixed points.

```
tree                       :: Fix TreeF -> Tree
tree (In (LeafF c))        =  Leaf c
tree (In (NodeF x y))      =  Node (tree x) (tree y)

untree                :: Tree -> Fix TreeF
untree (Leaf c)       =  In (LeafF c)
untree (Node x y)     =  In (NodeF (untree x) (untree y))
```

`Nat` as a fixed point

```
data Nat       =  Zero | Succ Nat
data NatF a    =  ZeroF | SuccF a

nat                  :: Fix NatF -> Nat
nat (In ZeroF)       =  Zero
nat (In (SuccF n))   =  Succ (nat n)

unnat              :: Nat -> Fix NatF
unnat Zero         =  In ZeroF
unnat (Succ n)     =  In (SuccF (unnat n))
```

Limitations of data types as fixed points

Viewing a data type as a fixed point implies a number of limitations: the following classes of data types cannot be modelled anymore.

- Nested data types

```
data Fork a  =  ForkF a a
data Sequ a  =  EndS
             |  ZeroS (Sequ (Fork a))
             |  OneS a (Sequ (Fork a))
```

- Mutual recursive data types:

```
data Rose a    =  Rose a (Forest a)
data Forest a  =  FNil | FCons (Rose a) (Forest a)
```

Locations on trees

A location on on a tree is a tree, together with a context.

```
data TreeF a  =  LeafF Char | NodeF a a

type Loc_Tree = (Fix TreeF, Context_Tree)

data Context_Tree  =  Top
                   |  LNode Context_Tree Tree
                   |  RNode Tree Context_Tree
```

A context of a tree is either the top context, or it is a description from the top to the current position. The complete tree is recovered as follows:

```
cTree                    :: Loc_Tree -> Tree
cTree (t,Top)            =  t
cTree (t,LNode c t')     =  cTree (Node t t',c)
cTree (t,RNode t' c)     =  cTree (Node t' t,c)
```

Locations on natural numbers

A location on on a natural number is defined as follows.

```
data NatF a    =  ZeroF | SuccF a

type Loc_Nat = (Fix NatF, Context_Nat)

data Context_Nat  =  Top
                  |  DSucc Context_Nat
```

Note that the context of a natural number is a natural number again!

Generic locations

To define generic locations we use a type-indexed data type.

```
type LOC{| f :: * -> * |}     =
  (Fix f,CONTEXT {| f |} (Fix f))
type CONTEXT{| f :: * -> * |} r   =
  Fix (LMaybe (CTX {| f |} r))

data LMaybe f a = LNothing | LJust (f a)
```

I will write `Nothing` (`Just`) for `LNothing` (`LJust`).

A type-indexed data type for contexts

The type `CTX {| f |}` is the *derivative* of the type `f`:

$$\begin{aligned} const' &= 0 \\ (x+y)' &= x'+y' \\ (x*y)' &= x'*y+x*y' \end{aligned}$$

In Generic Haskell:

```
dependency CTX <- GID CTX

type CTX{| Unit |}               = CTXUnit Void
type CTX{| Char |}               = CTXChar Void
type CTX{| :+:  |} iA cA iB cB   = CTXSum  (Sum cA cB)
type CTX{| :*:  |} iA cA iB cB   = CTXProd (Sum (Prod cA iB)
                                                (Prod iA cB))
type CTX{| Con  |} iA cA         = CTXCon  cA
type CTX{| Label |} iA cA        = CTXLab  cA
```

Dependencies on type-indexed data types

Just as on type-indexed functions, we can specify dependencies on type-indexed data types. The line

```
dependency CTX <- GID CTX
```

says that the type-indexed data type `CTX` depends on both the identity type-indexed data type, and itself.

The identity type-indexed data type

```
type GID{| Unit |}      =  GIDUnit Unit
type GID{| Char |}      =  GIDChar Char
type GID{| :+: |} a b   =  GIDSum  (Sum a b)
type GID{| :*: |} a b   =  GIDProd (Prod a b)
type GID{| Con |} a     =  GIDCon  (Con a)
type GID{| Label |} a   =  GIDLabel (Label a)
```

To the identity type-indexed data type

```
type MkId  {[ * ]}       t  =  t -> GID {| t |}
type MkId  {[ k -> l ]} t  =
  forall u. MkId {[ k ]} u -> MkId {[ l ]} (t u)

mkid {| t :: k |}                 :: MkId {[ k ]} t
mkid {| Unit |} t                 =  GIDUnit t
mkid {| Int |}  t                 =  GIDInt  t
mkid {| Char |} t                 =  GIDChar t
mkid {| :+: |}  mA mB (Inl x)     =  GIDSum  (Inl (mA x))
mkid {| :+: |}  mA mB (Inr y)     =  GIDSum  (Inr (mB y))
mkid {| :*: |}  mA mB (x :*: y)   =  GIDProd (mA x :*: mB y)
mkid {| Con c |}  mA  (Con t)     =  GIDCon  (Con (mA t))
mkid {| Label l |} mA (Label t)   =  GIDLabel (Label (mA t))
```

Examples of function down

Function down on trees takes a location, and goes down to the leftmost tree child of a node if the current selection is a node, and does nothing otherwise:

```
down_Tree                 :: Loc_Tree -> Loc_Tree
down_Tree (Leaf a,c)      =  (Leaf a,c)
down_Tree (Node l r,c)    =  (l,LNode c r)
```

On Nat, function down is a variant of the predecessor function.

```
down_Nat               :: Loc_Nat -> Loc_Nat
down_Nat (Zero,c)      =  (Zero,c)
down_Nat (Succ n,c)    =  (n,DSucc c)
```

Function `down`

The generic function `down` analyses the current focus of attention, and moves down if possible.

```
down{|f :: * -> *|}  :: LOC{| f |} -> LOC{| f |}
down{| f |} (t,c)     =
  case first{| f |} (out t) c of
    Nothing        ->  (t,c)
    Just (t',c')   ->  (t', In (Just c'))
```

where function `first` is a type-indexed function that possibly returns the leftmost recursive child of a node, together with the context of the selected child.

Function `first`

```
first{| f :: * -> * |}  ::
   f (Fix f) -> c -> Maybe (Fix f, CTX{| f |} (Fix f) c)
first{| f |} x c  =  first'{| f |} first'Rec mkid x c

first'Rec t c  =  Just (t,c)

dependency first' <- first' mkid

type First{[  *   ]} t a c  =
  t -> c -> Maybe (a, CTX{| t |})
type First{[ k->l ]} t a c  =
  forall u. First{[ k ]} u a c -> MkId{[ k ]} u ->
            First{[ l ]} (t u) a c
```

Function `first’`

```
first’{| t :: k |} :: forall a c. First {[ k ]} t a c
first’{| Unit |}                t          c = Nothing
first’{| Char |}                t          c = Nothing
first’{| :+: |} fA mA fB mB (Inl x)    c =
  do (t,cx) <- fA x c; return (t,CTXSum (Inl cx))
first’{| :+: |} fA mA fB mB (Inr y)    c =
  do (t,cy) <- fB y c; return (t,CTXSum (Inr cy))
first’{| :*: |} fA mA fB mB (x :*: y) c =
  (do (t,cx) <- fA x c
      return (t,CTXProd (Inl (cx:*:mB y))))
  ‘mplus‘
  (do (t,cy) <- fB y c
      return (t,CTXProd (Inr (mA x:*:cy))))
first’ {| Con d |} fA mA (Con t) c       =
  do (t,cx) <- fA t c;  return (t,CTXCon cx)
```

A property of function `down`

Function `down` should satisfy the following property.

```
forall l . down{|f|} l /= l  =>  (up{|f|} . down{|f|}) l = l
```

where function up goes up in a tree. So first going down the tree and then up again is the identity function on locations in which it is possible to go down.

Examples of function up

Function up on trees takes a location, and goes up to the the parent of the current selection if the current selection is not the complete tree.

```
up_Tree                    :: Loc_Tree -> Loc_Tree
up_Tree (t,Top)             =  (t,Top)
up_Tree (t,LNode c tr)      =  (Node t tr,c)
up_Tree (t,RNode tr c)      =  (Node tr t,c)
```

On Nat, function up is a variant of the successor function.

```
up_Nat                 :: Loc_Nat -> Loc_Nat
up_Nat (n,Top)          =  (n,Top)
up_Nat (n,DSucc c)      =  (Succ n,c)
```

Function `up`

The generic function `up` analyses the context of the current focus of attention, and moves up if possible.

```
up{|f :: * -> *|}  :: LOC{| f |} -> LOC{| f |}
up{| f |} (t,c)     =
  case out c of
    Nothing  ->  (t,c)
    Just c'  ->  fromJust $
                 do ft <- insert{|f|} c' t
                    c'' <- extract{|f|} c'
                    return (In ft,c'')
```

where function `insert` is a type-indexed function that takes a context and a tree, and inserts the tree in the current focus of attention, and function `extract` extracts the context of the parent of the current focus of attention.

I will just define function `extract`.

Function extract

```
extract{| f :: * -> * |}  :: CTX{| f |} t c -> Maybe c
extract{| f |} c           =  extract'{|f|} Just c

type Extract{[  *   ]} t a  =  CTX{| t |} -> Maybe a
type Extract{[ k->l ]} t a  =
  forall u . Extract{[ k ]} u a -> Extract{[ l ]} (t u) a

extract'{| t :: k |} :: forall a. Extract{[ k ]} t a
extract'{| Unit |}    c                             = Nothing
extract'{| Char |}    c                             = Nothing
extract'{| :+: |} eA eB (CTXSum (Inl cx))           = eA cx
extract'{| :+: |} eA eB (CTXSum (Inr cy))           = eB cy
extract'{| :*: |} eA eB (CTXProd (Inl (cx :*: y)))  = eA cx
extract'{| :*: |} eA eB (CTXProd (Inr (x :*: cy)))  = eB cy
extract'{| Con c |} eA    (CTXCon cx)               = eA cx
extract'{| Label l |} eA  (CTXLab cx)               = eA cx
```

Conclusions and future work

I have shown:

- How you can define the zipper datastructure as a collection of generic definitions, with bot type-indexed data types and type-indexed functions.
- How dependencies are used in generic definitions.

Future work:

- Use the zipper in a real structured editor.