The Zipper

Ralf Hinze, Johan Jeuring

August ??, 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.
- Give some possible generic programming projects.

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 datatype 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 (Maybe (CTX {| f |} r))
```

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{[  *   ]} t  =  t -> t
type GId{[ k->l ]} t  =
  forall a. GId{[ k ]} a -> GId{[ l ]} (t a)

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)
```

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 id 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.

Generic Programming projects

Here are some research problems on Generic Programming:

- Applications: Develop a compiler tool such as a debugger as a generic program.
- Theory: Develop heuristics to generate generic functions from a number of instances on some data types.
- Language: Grammar analyses can be viewed as type-indexed functions. What features do we need to be able to express grammar analyses as generic programs?