J. Functional Programming **1** (1): 1–000, January 1993 © 1993 Cambridge University Press

FUNCTIONAL PEARLS

The Zipper

Gérard Huet
INRIA Rocquencourt, France

Capsule Review

Almost every programmer has faced the problem of representing a tree together with a subtree that is the focus of attention, where that focus may move left, right, up, or down the tree. The Zipper is Huet's nifty name for a nifty data structure which fulfills this need. I wish I had known of it when I faced this task, because the solution I came up with was not quite so efficient or elegant as the Zipper.

Introduction

The main drawback to the purely applicative paradigm of programming is that many efficient algorithms use destructive operations in data structures such as bit vectors or character arrays or other mutable hierarchical classification structures, which are not immediately modeled as purely applicative data structures. A well known solution to this problem is called *functional arrays*(Paulson, 1991). For trees, this amounts to modifying an occurrence in a tree non-destructively by copying its *path* from the root of the tree. This is considered tolerable when the data structure is just an object local to some algorithm, the cost being logarithmic compared to the naive solution which copies all the tree. But when the data structure represents some global context, such as the buffer of a text editor or the data base of axioms and lemmas in a proof system, this technique is prohibitive. In the current note, we explain a simple solution, where tree editing is completely local, the handle on the data being not the original root of the tree, but rather the current position in the tree.

The basic idea is simple: the tree is turned inside-out like a returned glove, pointers from the root to the current position being reversed in a *path structure*. The current *location* holds both the downward current subtree, and the upward path. All navigation and modification primitives operate on the location structure. Going up and down in the structure is analogous to closing and opening a zipper in a piece of clothing, whence the name.

The author coined this data-type when designing the core of a structured editor for use as structure manager for a proof assistant. This simple idea must have been invented at numerous occasions by creative programmers, and the only justification for presenting what ought to be folklore is that it does not appear to have been published, or even to be well-known.

1 The Zipper data structure

There are many variations on the basic idea. First let us present a version which pertains to trees with variadic arity anonymous tree nodes, and tree leaves injecting values from an unspecified *item* type.

1.1 Trees, paths and locations

We assume a type parameter `item` of the elements we want to manipulate hierarchically; the tree structure is just hierarchical lists grouping trees in a section. For instance, in the UNIX file system, items would be files and sections would be directories; in a text editor items would be characters, and two levels of sections would represent the buffer as a list of lines and lines as lists of characters. Generalising this to any level, we would get a notion of hierarchical Turing machine, where a tape position may contain either a symbol or a tape from a lower level.

All algorithms presented here are written concretely in the programming language OCaml(Leroy et al., 1996). This code is translatable easily in any programming language, functional or not, lazy or not.

```
type tree =
    Item of item
  | Section of tree list;;
```

We now consider a `path` in a `tree`:

```
type path =
    Top
  | Node of tree list * path * tree list;;
```

A `path` is like a zipper, allowing one to rip the tree structure down to a certain location. A `Node(l,p,r)` contains its list `l` of elder siblings (starting with the eldest), its father path `p`, and its list of younger siblings (starting with the youngest).

Note: a tree presented by a path has sibling trees, uncle trees, great-uncle trees, etc, but its father is a path, not a tree like in usual graph editors.

A `location` in the tree adresses a subtree, together with its `path`.

```
type location = Loc of tree * path;;
```

A `location` consists of a distinguished `tree`, the current focus of attention, and its `path`, representing its surrounding context. Remark that a location does *not* correspond to an occurrence in the tree, as assumed for instance in term rewriting theory (Huet, 1980) or in tree editors (Donzeau-Gouge et al., 1984). It is rather a pointer to the arc linking the designated subtree to the surrouding context.

Example. Assume we consider the parse tree of arithmetic expressions, with string items. The expression $a \times b + c \times d$ parses as the tree:

```
Section[Section[Item "a"; Item "*"; Item "b"];
        Item "+";
        Section[Item "c"; Item "*"; Item "d"]];;
```

The location of the second multiplication sign in the tree is:

```
Loc(Item "*",
    Node([Item "c"],
         Node([Item "+"; Section [Item "a"; Item "*"; Item "b"]],
              Top,
              []),
         [Item "d"]))
```

1.2 Navigation primitives in trees

```
let go_left (Loc(t,p)) = match p with
    Top -> failwith "left of top"
  | Node(l::left,up,right) -> Loc(l,Node(left,up,t::right))
  | Node([],up,right) -> failwith "left of first";;

let go_right (Loc(t,p)) = match p with
    Top -> failwith "right of top"
  | Node(left,up,r::right) -> Loc(r,Node(t::left,up,right))
  | _ -> failwith "right of last";;

let go_up (Loc(t,p)) = match p with
    Top -> failwith "up of top"
  | Node(left,up,right) -> Loc(Section((rev left) @ (t::right)),up);;

let go_down (Loc(t,p)) = match t with
    Item(_) -> failwith "down of item"
  | Section(t1::trees) -> Loc(t1,Node([],p,trees))
  | _ -> failwith "down of empty";;
```

Note: all navigation primitives take constant time, except `go_up`, which is proportional to the 'juniority' of the current term `list_length(left)`.

We may program with these primitives the access to the nth son of the current tree.

```
let nth loc = nthrec
  where rec nthrec = function
    1 -> go_down(loc)
  | n -> if n>0 then go_right(nthrec (n-1))
                else failwith "nth expects a positive integer";;
```

1.3 Changes, insertions and deletions

We may mutate the structure at the current location as a local operation:

```
let change (Loc(_,p)) t = Loc(t,p);;
```

Insertion to the left or to the right is natural and cheap.

```
let insert_right (Loc(t,p)) r = match p with
    Top -> failwith "insert of top"
  | Node(left,up,right) -> Loc(t,Node(left,up,r::right));;

let insert_left (Loc(t,p)) l = match p with
    Top -> failwith "insert of top"
  | Node(left,up,right) -> Loc(t,Node(l::left,up,right));;

let insert_down (Loc(t,p)) t1 = match t with
    Item(_) -> failwith "down of item"
  | Section(sons) -> Loc(t1,Node([],p,sons));;
```

We may also want to implement a deletion primitive. We may choose to move right, if possible, otherwise left, and up in case of an empty list.

```
let delete (Loc(_,p)) = match p with
    Top -> failwith "delete of top"
  | Node(left,up,r::right) -> Loc(r,Node(left,up,right))
  | Node(l::left,up,[]) -> Loc(l,Node(left,up,[]))
  | Node([],up,[]) -> Loc(Section[],up);;
```

We remark that `delete` is not such a simple operation.

We believe that the set of datatypes and operations above is adequate for programming the kernel of a structure editor in an applicative albeit efficient manner.

2 Variations on the basic idea

2.1 Scars

When an algorithm has frequent operations which necessitate going up in the tree, and down again at the same position, it is a loss of time (and space, and garbage-collecting time, etc) to close the sections in the meantime. It may be advantageous to leave "scars" in the structure allowing direct access to the memorized visited positions. Thus we replace the (non-empty) sections by triples memorizing a tree and its siblings:

```
type memo_tree =
    Item of item
  | Siblings of memo_tree list * memo_tree * memo_tree list;;

type memo_path =
    Top
  | Node of memo_tree list * memo_path * memo_tree list;;

type memo_location = Loc of memo_tree * memo_path;;
```

We show the simplified up and down operations on these new structures:

```
let go_up_memo (Loc(t,p)) = match p with
    Top -> failwith "up of top"
  | Node(left,p',right) -> Loc(Siblings(left,t,right),p');;

let go_down_memo (Loc(t,p)) = match t with
    Item(_) -> failwith "down of item"
  | Siblings(left,t',right) -> Loc(t',Node(left,p,right));;
```

We leave it to the reader to adapt other primitives.

2.2 First-order terms

So far, our structures are completely untyped, our tree nodes are not even labelled. We have a kind of structured editor à la LISP, but oriented more toward “splicing” operations than the usual `rplaca` and `rplacd` primitives.

If we want to implement a tree-manipulation editor for abstract-syntax trees, we have to label our tree nodes with operator names. If we use items for this purpose, this suggests the usual LISP encoding of first-order terms: $F(T_1, ..., T_n)$ being coded as the tree `Section[Item(F); T1; ... Tn]`. A dual solution is suggested by combinatory logic, where the comb-like structure respects the application ordering: `[Tn; ... T1; Item(F)]`. Neither of these solutions respects arity however.

We shall not pursue details of such generic variations any more, but rather consider how to adapt the idea to a *specific* given signature of operators given with their arities, in such a way that tree edition maintains well-formedness of the tree according to arities.

Basically, to each constructor F of the signature with arity n we associate n path operators $Node(F, i)$, with $1 \leq i \leq n$, each of arity n, used when going down the i-th subtree of an F-term. More precisely, $Node(F, i)$ has one path argument and $n - 1$ tree arguments holding the current siblings.

We show for instance the structure corresponding to binary trees:

```
type binary_tree =
    Nil
  | Cons of binary_tree * binary_tree;;

type binary_path =
    Top
  | Left of binary_path * binary_tree
  | Right of binary_tree * binary_path;;

type binary_location = Loc of binary_tree * binary_path;;

let change (Loc(_,p)) t = Loc(t,p);;
let go_left (Loc(t,p)) = match p with
    Top -> failwith "left of top"
  | Left(father,right) -> failwith "left of Left"
  | Right(left,father) -> Loc(left,Left(father,t));;

let go_right (Loc(t,p)) = match p with
    Top -> failwith "right of top"
  | Left(father,right) -> Loc(right,Right(t,father))
  | Right(left,father) -> failwith "right of Right";;

let go_up (Loc(t,p)) = match p with
    Top -> failwith "up of top"
  | Left(father,right) -> Loc(Cons(t,right),father)
  | Right(left,father) -> Loc(Cons(left,t),father);;

let go_first (Loc(t,p)) = match t with
    Nil -> failwith "first of Nil"
  | Cons(left,right) -> Loc(left,Left(p,right));;

let go_second (Loc(t,p)) = match t with
    Nil -> failwith "second of Nil"
  | Cons(left,right) -> Loc(right,Right(left,p));;
```

Efficient destructive algorithms on binary trees may be programmed with these completely applicative primitives, which all use constant time, since they all reduce to local pointer manipulation.

References

V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang. "Programming Environments based on Structured Editors: The MENTOR Experience." In **Interactive Programming Environments**. Eds. Barstow D., Shrobe H. and Sandewall E., McGraw Hill, 1984, pp. 128-140.

G. Huet. "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems." J. Assoc. Comp. Mach. **27,4** (1980) 797–821.

X. Leroy, D. Rémy, J. Vouillon. "The Objective Caml system, documentation and user's manual - release 1.02." INRIA, oct. 1996. Available on `ftp.inria.fr:INRIA/Projects/cristal`.

L. C. Paulson. "ML for the working programmer." Cambridge University Press, 1991.