# Representation Issues for Symbolic Computation


**Gérard Huet**

**INRIA**


35 years of Automath, April 2002

# A few design issues

When one designs a platform for symbolic computations, be it for logical frameworks, for computational mathematics environments, or for computational linguistic toolkits, many common issues arise.

We shall discuss in this talk two specific issues:

- Local manipulation of focused data
- Sharing

for which we shall propose solutions, and we shall evoke numerous other issues which are still the object of debate, and perhaps research.

# Basics: lists vs stacks

```
value l5 = [1; 2; 3; 4; 5];

value s5 = [5; 4; 3; 2; 1];

value rec unstack l s =
  match l with
    [ [] -> a
    | [h::t] -> unstack t [h::s]
    ];

value rev l = unstack l [];

value state3 = ([3; 2; 1],[4; 5]);
```

# Turing machines, Emacs, and Zippers

First presentation at FLoC'96. Published as:
G. Huet. The Zipper. J. Functional Programming 7,5 (1997), 549-554.

Large scale implementations in syntax editors within computational linguistics platforms:

- G. Huet. Lexical morphisms.
- A. Ranta. Grammatical frameworks.

# Contexts as zippers

```
type tree = [ Tree of arcs ]
and arcs = list tree;

type tree_zipper =
  [ Top
  | Zip of (arcs * tree_zipper * arcs)
  ];

type focused_tree = (tree_zipper * tree);
```

A focused tree is a tree with a focus point of interest, i.e. a tree and a stacked context.

# Operations on focused trees

```
value down (z,t) = match t with
   [ Tree(arcs) -> match arcs with
     [ [] -> raise (Failure "down")
     | [hd::tl] -> (Zip([],z,tl),hd)
     ]
   ];

value up (z,t) = match z with
   [ Top -> raise (Failure "up")
   | Zip(l,u,r) -> (u, Tree(unstack l [t::r]))
   ];
```

# More operations on focused trees

```
value left (z,t) = match z with
  [ Top -> raise (Failure "left")
  | Zip(l,u,r) -> match l with
    [ [] -> raise (Failure "left")
    | [elder::rest] -> (Zip(elders,u,[t::r]),rest)
    ]
  ];

value right (z,t) = match z with
  [ Top -> raise (Failure "right")
  | Zip(l,u,r) -> match r with
    [ [] -> raise (Failure "right")
    | [young::rest] -> (Zip([t::l],u,rest),young)
    ]
  ];
```

## Applicative updating

```
value del_l (z,_) = match z with
   [ Top -> raise (Failure "del_l")
   | Zip(l,u,r) -> match l with
      [ [] -> raise (Failure "del_l")
      | [elder::rest] -> (Zip(rest,u,r),elder)
      ]
   ];

value replace (z,_) t = (z,t);
```

## Restoring the symmetry ?

```
type tree2 = [ Parent of (arcs2 * tree2 * arcs2)
             | Void
             ]
and arcs2 = list tree2;
```

Cf. ternary search trees (Bentley, Sedgewick).

Note: often proof engines implement the context as a left-only zipper, imposing (too) strong sequentiality requirements on the proof search algorithm..

## Points of view about focused structures

- Manipulation of focused data is local
- Redundant representation - efficiency
- The Interaction Combinators Paradigm

Important epistemological remark. The computational cost of a notation is relevant to the difficulty of its understanding. This point has been overlooked in mainstream mathematics. The same mistake is being made even in the teaching of functional programming.

## Points of view about zippers

- Applicative contexts
- As opposed to $\Omega$-terms
- Approximants of coinductive structures
- Information ordering is substructural
- Generalisations of D-lists
- Linear maps

## Zipper application and composition

A zipper `z` may be thought of as the function from tree `t` to tree `zip_up z t`, with:

```
value rec zip_up z t = match z with
  [ Top -> t
  | Zip(l,up,r) -> zip_up up (Tree(unstack l [t::r]))
  ];
```

Applying it to `t` substitutes `t` in the place holder represented by `z`. If we substitute another zipper, we obtain zipper composition:

```
value rec zip_unstack z1 z2 = match z1 with
  [ Top -> z2
  | Zip(l,z,r) -> zip_unstack z (Zip(l,z2,r))
  ];


value zip_rev z = zip_unstack z Top;
```

## The Zipper Category of linear tree maps

And now composition is similar to concatenation of lists:

```
value compose z1 z2 =
  zip_unstack (zip_rev z2) z1;
```

It is easy to show that `Top` is an identity on the left and on the right for composition, and that composition is associative. Thus we get a category, whose objects are trees and morphisms are zippers, which we call the Zipper category of linear tree maps.

## Tree splicing

Tree splicing, or *adjunction* in the terminology of Tree Adjoint Grammars, is very naturally expressible in this framework. Indeed, what is called a rooted tree in this tradition is here directly expressed as a zipper `zroot`, and adjunction at a tree occurrence is prepared by decomposing this tree at the given occurrence as a focused tree `(z,t)`. Now the adjunction of `zroot` at this occurrence is:

```
value splice_down (z,t) zroot = (compose z zroot, t);
```

if the focus of attention stays at the subtree `t`, or

```
value splice_up (z,t) zroot = (z, zip_up zroot t);
```

if we want the focus of attention to stay at the adjunction locus. These two points of view lead to equivalent structures, in the sense of tree identity modulo focusing:

```
value equiv (z,t) (z',t') = (zip_up z t = zip_up z' t');
```

## Links with Linear Logic

A tree focused at occurrence $u$ is like a box

$$\frac{T}{T \multimap T \quad T}$$

or, in a multisorted algebra:

$$\frac{B}{A \multimap B \quad A}$$

The zipper has type $Z = T \multimap T$ and the focused tree has type $S = Z \otimes T = T \multimap T \otimes T$.

So by computation we retrieve the term from the zipper-context and the subterm, whereas navigation just "swims in the isomorphisms".

# Linearization

The important point is that if $T$ is an algebraic inductive type, then so is $Z$. We get from the defining equation $T = F(T)$ where $F$ is the defining functor to the defining equation $Z = G(Z)$ where $G = Lin_T(F)$ with $Lin_T$ the natural transformation which linearizes $F$.

For instance with binary trees: $F(T) = 1 \oplus (T \otimes T)$ we get $G(Z) = 1 \oplus (T \otimes Z) \oplus (Z \otimes T)$.

But with lists, we have $F(T) = 1 \oplus T$ already linear, so $G = F$ and this is why we confuse lists and stacks! And now we understand that `unstack` is actually `apply` with the finer typing `stack`$=$`list`$\multimap$`list`.

Stacks are approximations of streams, and more generally over algebraic types $Z = T \multimap T$ is the finite part of the co-inductive $T^\perp$.

## The Share Functor

```
module Share : functor (Algebra:sig type domain = 'a;
                                    value size: int; end) ->
sig value share: Algebra.domain->int->Algebra.domain; end;
```

That is, *Share* takes as argument a module *Algebra* providing a type *domain* and an integer value *size*, and it defines a value *share* of the stated type. We assume that the elements from the domain are presented with an integer key bounded by Algebra.size. That is, `share x k` will assume as precondition that $0 \leq k < Max$ with $Max =$`Algebra.size`.

We shall construct the sharing map with the help of a hash table, made up of buckets $(k, [e_1; e_2; ...e_n])$ where each element $e_i$ has key $k$.

## Memoizing

```
type bucket = list Algebra.domain;
```

```
value memo = Array.create Algebra.size ([] : bucket);
```

We shall use a service function `search`, such that *search e l* returns the first $y$ in $l$ such that $y = e$ or or else raises the exception `Not_found`.

```
value search e = List.find (fun x -> x=e);
```

# The share function

```
value share element key =
  let bucket = memo.(key) in
  try search element bucket with
    [ Not_found ->
      do {memo.(key):=[element::bucket]; element}
    ];
```

Sharing is just recalling!

## Compressing trees as dags

We may for instance instantiate *Share* on the algebra of trees, with a size *hash_max* depending on the application:

```
module Dag = Share (struct type domain=tree;
                           value size=hash_max; end);
```

And now we compress a trie into a minimal dag using *share* by a simple bottom-up traversal, where the key is computed along by hashing. For this we define a general bottom-up traversal function, which applies a parametric `lookup` function to every node and its associated key.

## Dynamic programming

Bottom-up traversing with inductive hash-code computation.

```
(* linear hash-code parameters *)
value hash1 key index sum = sum + index*key
and hash arcs = arcs mod hash_max;

value traverse lookup = travel
 where rec travel = fun
  [ Tree(arcs) ->
    let f (tries,index,span) t =
       let (t0,k) = travel t
       in  ([t0::tries],index+1,hash1 k index span)
    in let (arcs0,_,span) = List.fold_left f ([],1,0) arcs
       in let key = hash span
          in (lookup (Tree(rev arcs0)) key, key)
  ];
```

## Compressing a tree as a dag

Now, compressing a tree optimally as a minimal dag is simply effected by a sharing traversal:

```
value compress = traverse Dag.share;

value minimize tree = let (dag,_) = compress tree in dag;
```

## Advantages and extensions

Hashing keys and size is on the client side : we do not delegate hashing to Share, which is just an associative memory. This has two advantages:

- The computation is fully linear
- It is adapted to the statistics of the data

Extension : Auto-sharing types (controlled hash-consing). Suggests a monad of shared hashed structures accommodating entropy of the data.

# Other issues

- Bound variables
- Existential variables
- Definitions
- Opacity
- Modules
- Libraries
- Vernacular
- Automation
- Cooperation