Linear Contexts and the Sharing Functor: Techniques for Symbolic Computation

Gérard Huet

INRIA-Rocquencourt

Abstract

We present in this paper two design issues concerning fundamental representation structures for symbolic and logic computations. The first one concerns structured editing, or more generally the possibly destructive update of tree-like data-structures of inductive types. Instead of the standard implementation of mutable data structures containing references, we advocate the *zipper* technology, fully applicative. This may be considered a disciplined use of pointer reversal techniques. We argue that zippers, i.e. unary *contexts* generalizing stacks, are concrete representations of linear functions on algebraic data types. The second method is a uniform *sharing* functor, which is a variation on the traditional technique of *hashing*, but controling the indexing function on the client side rather than on the server side, which allows the fine-tuning of bucket balancing, taking into account specific statistical properties of the application data. Such techniques are of general interest for symbolic computation applications such as structure editors, proof assistants, algebraic computation systems, and computational linguistics platforms.

For 35 years of Automath.

Introduction

We present a few basic programming techniques which are of general use in the design of symbolic computation systems, be it for logical frameworks, for computational mathematics environments, or for computational linguistic toolkits. Our methodology has a mixture of algebraic and algorithmic flavors. The main idea is to keep applicative representations of logical structure, manipulated through combinators which keep operations local. Data structures are split at a point of focus into a *context* and a *substructure* with opposite polarities. Elementary combinators permit to navigate in the structure, changing the focal point, but also to modify the structure by local editing, in the spirit of interaction combinators [13].

Another consideration is to optimise storage use by keeping compact representations with maximum sharing. A general *sharing functor* is exhibited, and compared to the traditional hash-code table technology. This technique generalises dynamic programming techniques, such as the minimisation of deterministic finite-state automata.

We shall use as *meta language* for the description of our algorithms a Pidgin version of the functional language ML. Actually, our Pidgin ML is nothing else than a core subset of Objective Caml [18], under the so-called revised syntax [17], and thus all algorithms appearing below are directly executable and are actually part of the Zen toolkit for computational linguistics under development by the author [11].

1 Top-down structures vs bottom-up structures

We understand well top-down (inductive) structures. They are the representations of initial algebra values. For instance, the structure `bool` has two constant constructors, the booleans `True` and `False`. The polymorphic structure `list 't` admits two constructors, the empty list `[]` and the list constructor consing a value `x` of (polymorphic) type `'a` to a homogeneous list `l` of type `list 'a` to form `[a::l]` of type `list 'a`.

Bottom-up structures are useful for creating, editing, traversing and changing top-down structures in a local but applicative manner. They are sometimes called computation contexts. We shall call them *zippers*, following [9].

Top-down structures are the finite elements inhabiting inductively defined types. Bottom-up structures are also finite, but they permit the progressive definition of (potentially infinite) values of co-inductive types. They permit incremental navigation and modification of very general data types values. We shall also see that they model linear structural functions, in the sense of linear logic.

Finally, bottom-up computing is the right way to build shared structures in an applicative fashion, opening the optimisation path from trees to dags. Binding algebras (λ-calculus expressions for inductive values and Böhm trees [8] for the co-inductive ones) may be defined by either de Bruijn indices [4] or higher-order abstract syntax, and more general graph structures may be represented by spanning trees decorated with virtual adresses, so we see no reason to keep explicit references and pointer objects, with all the problems they are liable for, and we shall stick to purely applicative programming.

1.1 Lists, stacks and queues

Queues are first-in first-out sequences (top-down) whereas stacks are last-in first-out sequences (bottom-up). They are not clearly distinguished in usual programming, because the underlying data structure is the same : the queue `[x1; x2; ... xn]` may be reversed into the stack `[xn ...; x2; x1]` which is of the same *type* list. So we cannot expect to distinguish them with the type discipline of ML. At best by declaring:
`type stack 'a = list 'a and queue 'a = list 'a;`
we may use type annotations to document whether a given list is used by a function in the rôle of a queue or of a stack. But such *intentions* are not enforced by ML's type system, which just uses freely the type declaration above as an equivalence. So we have to check these intentions carefully, if we want our values to come in the right order. But we certainly wish to distinguish queues and stacks, since stacks are built and analysed in unit time, whereas adding a new element to a queue is proportional to its length.

Of course we could define a new type `queue` with different constructors than the list one, but this would hardly solve the problem. Even though the two types would be isomorphic, ML would not allow their coercion, and since one's list is often someone else's stack this would incur undue copying costs. Furthermore it would not really account for the queue data structure, which demands a more complicated implementation with references. So we have to live in a world where lists are really stacks (since the list cons primitive is pushing on a stack), whereas mathematical notation for sequences rather follows the queue discipline, and keep proper track of reversals.

For instance, here is a typical example of stack use:

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

In `unstack`, `s` is an accumulator stack, where values are listed in the opposite order as they are in list `l`. Indeed, we may define the reverse operation on lists as:

```
value rev l = unstack l [];
```

In the standard Ocaml's library, `unstack` is called `rev_append`. It is efficient, since it is *tail recursive*: no intermediate values of computation need to be kept on the recursion stack, and the recursion is executed as a mere jump. It is much more efficient, if some list `l1` is kept in its reversed stack form `s1`, to obtain the result of appending `l1` to `l2` by calling `rev_append s1 l2` than to call `append l1 l2`, which amounts to reversing first `l1` into `s1`, and then doing the same computation. Similarly, the library defines a function `rev_map` which is more efficient than `map`, if one keeps in mind that its result is in the stack order. But no real discipline of these library functions is really enforced. Two iterators are given, `fold_left` and `fold_right`, which gives no encouragement to use the first one, which is tail recursive, and no warning that its results are accumulated in the stack order rather than in the queue order as the second one does. And complexity analysis is not of any use, since all the functions are linear in the sense of being in the complexity class $O(n)$, with n the length of the traversed list. So this distinction between lists and stacks is kept fuzzy, as a programming technique learnt "hands on" so to speak, but not really worth much discussion. Students (and even seasoned programmers) will occasionally get results in the wrong order, and they will sprinkle their code with explicit reverse operations here and there until it works; professors will teach assertions and invariants and verification conditions which will prove that indeed the intended values are produced, but here the type system of ML is of no use, since it classifies `l` and `rev l` as consistent lists.

Here we want to make this distinction precise, favor local operations, and delay as much as possible any reversal. For instance, if some list `l1` is kept in its reversed stack form `s1`, and we wish to append list `l2` to it, the best is to just wait and keep the pair `(s1,l2)` as the state of computation where we have `l2` *in the context* `s1`. In this computation state, we may finish the construction of the result `l` of appending `l1` to `l2` by "zipping up" `l1` by executing `unstack s1 l2`, or we may choose rather to "zip down" `l2` by executing `unstack l2 s1` to get the stack context value `rev l`. But we may also consider that the computation state `(s1,l2)` represents `l` locally accessed as its prefix `l1` stacked in context value `s1` followed by its suffix `l2`. And it is very easy to insert as this point a new element `x`, either stacked upwards in state `([x::s1],l2)`, or consed downwards in state `(s1,[x::l2])`.

Once this intentional programming methodology of representing focused structures as pairs `(context,substructure)` is clear, it is very easy to understand the generalisation to zippers, which are to general tree structures what stacks are to lists, i.e. upside-down access representations of (unary) contexts.

1.2 Contexts as Ω-terms

It is usual in denotational semantics to represent contexts as Ω-terms: you add one special constant Ω to the signature of your structure, and you consider terms with one "hole" filled with Ω, playing a role of place holder. Ω is some syntactic analogue of the undefined value $\perp$, and increasing your context by substituting some context or term to Ω is increasing in the corresponding domain ordering. But this solution is unsatisfactory for several reasons. First of all, in order to increase your context (stacking) you have computationally to look for the place holder. Second, there may be several Ωs in an Ω-term (or none), so you have to define precisely substitution and state what happens to non-linear terms, etc. When you actually substitute, you then rebuild the whole context term, or at least its spine.

It is true that these operations are cheap if classical mathematics is the meta-language, because the computational cost associated with the notation is not accounted for. But the whole approach is basically an incorrect point of view, and leads to constant worry about whether to quantify the lemmas over terms, over Ω-terms, or over linear Ω-terms. Just think of the computational contents of the approximations to the infinite stream of natural numbers as the sequence Ω, `[0::Ω]`, `[0::[1::Ω]]`, etc. This should be compared with representing the same approximations as the sequence of stack values `[]`, `[0]`, `[1;0]`, etc., where no special approximation apparatus is required, since the approximation ordering is just substructural. The correct point of view is to consider as meta-language a genuine programming language, where the computational costs somehow reflect the conceptual difficulty associated with the notation.

1.3 Contexts as zippers

Let us start with an example : ordered trees. We assume the mutual inductive types :

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

The tree zippers are the contexts of a place holder in the arcs, that is linked to its left siblings, right siblings, and parent context:

```
type tree_zipper =
  [ Top
```

```
  | Zip of (arcs * tree_zipper * arcs)
  ];
```

Let us model access paths in trees by sequences of natural numbers naming the successive arcs of a given node 1, 2, etc.

```
type access = list int
and domain = list access;
```

We usually define the domain of a tree as the set of accesses of its subterms:

```
value rec dom = fun
  [ Tree(arcs) ->
    let doms = map dom arcs in
    let f (n, d) dn = let ds = map (fun u -> [n::u]) dn in
                      (n+1,rev_append ds d) in
    let (_,d) = fold_left f (1,[[]]) doms in rev d
  ];
```

Thus, we get for instance:

```
value tree0=Tree [Tree [Tree []; Tree []]; Tree []];
dom(tree0);
[[]; [1]; [1; 1]; [1; 2]; [2]] : domain
```

Now if $u \in dom(t)$, we may decompose t at u into a term-in-context `at u t` and an Ω-term context `omega u t`. But this is needlessly complex and it is much better to replace the domain by the set of its reverses, that is by inducting on access stacks rather than access lists. Now if $rev(u) \in dom(t)$, we may zip-down t along u by changing focus, as follows:

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

value nth_context n = nthc n []
 where rec nthc n l = fun
   [ [] -> raise (Failure "out of domain")
   | [x::r] -> if n = 1 then (l,x,r) else nthc (n-1) [x::l] r
   ];

value rec enter u t = match u with
  [ [] -> ((Top,t) : focused_tree)
  | [n::l] -> let (z,t1) = enter l t in
              match t1 with
```

```
    [ Tree(arcs) -> let (l,t2,r)=nth_context n arcs in
                    (Zip(l,z,r),t2)
    ]
  ];
```

and now we may for instance navigate in `tree0` down to occurrence `[1;2]` by:

```
enter [2;1] tree0;

(Zip ([Tree []], Zip ([], Top, [Tree []]), []), Tree [])
  : focused_tree
```

1.4 Structured editing on focused trees

We shall not explicitly use these access stacks and the function `enter`; these access stacks are implicit from the zipper structure, and we shall navigate in focused trees one step at a time, using the following structure editor primitives on focused trees, of type `focused_tree -> focused_tree`.

```
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]))
   ];

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

value del_l (z,_) = match z with
```

```
   [ Top -> raise (Failure "del_l")
   | Zip(l,u,r) -> match l with
      [ [] -> raise (Failure "del_l")
      | [elder::elders] -> (Zip(elders,u,r),elder)
      ]
   ];

(* replace: focused_tree -> tree -> focused_tree *)
value replace (z,_) t = (z,t);
```

We skip operations `right` and `del_r`, symmetric to `left` and `del_l` respectively. Note how `replace` is a local operation, even though all our programming is applicative.

Remark. An alternate data type definition would emphasize the fact that a zipper is a stack of accesses in the tree:

```
type relatives = (arcs * arcs )
and zipper = list relatives;
```

Now the `relatives` value `(elders,youngers)` pairs the *stack* `elders` of elder siblings with the *queue* `youngers` of younger siblings of the corresponding ancestor in the zipper *stack* of ancestors of the current node. It is easy to adapt our algorithms to this alternate design, and many variations are possible. We prefer the original presentation where `Zip(left,up,right)` reflects visually the surrounding context, but this is essentially a question of taste.

1.5 Zipper operations

The editing operations above are operations on a finite tree represented at a focus point. But we may also define operations on zippers alone, which may be thought of as operations on a potentially infinite tree, actually on all trees, finite or infinite, having this initial context. That is, focused trees as pairs (context,structure) refer to finite elements (inductive values), whereas contexts may be seen as finite approximations to streams (co-inductive values), for instance generated by a possiblly non-terminating process. For example, here is an interpreter that takes a command to build progressively a zipper context:

```
type context_construction =
  [ Down | Left of tree | Right of tree ];
```

```
value build z = fun
  [ Down -> Zip([],z,[])
  | Left(t) -> match z with
     [ Top -> raise (Failure "build Left")
     | Zip(l,u,r) -> Zip([t::l],u,r)
     ]
  | Right(t) -> match z with
     [ Top -> raise (Failure "build Right")
     | Zip(l,u,r) -> Zip(l,u,[t::r])
     ]
  ];
```

But we could also add to our commands some destructive operations, to delete the left or right sibling, or to pop to the upper context.

1.6 Zippers as linear maps

We developed the idea that zippers were dual to trees in the sense that they may be used to represent the approximations to the co-inductive structures corresponding to trees as inductive structures. We shall now develop the idea that zippers may be seen as linear maps over trees, in the sense of linear logic. In the same way that a stack `st` may be thought of as a representation of the function which, given a list `l`, returns the list `unstack st l`, a zipper `z` may be thought of as the function which, given a tree `t`, returns the 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]))
  ];
```

Thus `zip_up` may be seen as a coercion between a zipper and a map from trees to trees, which is linear by construction, whereas a context as an Ω-term naturally leads to a potentially non-linear one, since there may be several occurrences of Ω (or none) in it.

Alternatively to computing `zip_up z t`, we could of course just build the focused tree `(z,t)`, which is a "lazy" representation which could be rolled in into `zip_up z t` if an actual term is needed later on.

Applying a zipper to a term is akin to substituting the term in the place holder represented by the zipper. If we substitute another zipper, we obtain zipper composition, as follows. First, we define the reverse of a zipper:

```
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;
```

And now composition is similar to concatenation of lists:

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

Alternatively, using a non-tail-recursive more direct definition:

```
value compose z1 z2 = comp z2
  where rec comp = fun [ Top -> z1
                       | Zip(l,z,r) ->  Zip(l,comp z,r)
                       ];
```

It is easy to check 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.

Alternatively, with the `zipper` type defined as `list relatives`, we would get composition by list concatenation.

We end this section by pointing out that tree splicing, or *adjunction* in the terminology of Tree Adjoint Grammars [12], 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 simply computed as:

```
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 occurrence. 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');
```

We remark that the redundancy in representation offered by equivalent focused trees, that is by the various ways of representing a tree locally at a focus point is a standard technique for optimising information processing. Such techniques are well-known for instance in hardware design, where redundant representations (introduced by Avizienis [16]) permit to keep the carry local in parallel adders. Redundant representations are also used to optimise algorithms for exact real arithmetic.

In the context of structure editing, a typical optimisation is to construct progressively a tree in a left-to-right top-down manner, keeping the focus at the frontier, so as to minimise the travel from one insertion to the next. This is what happens typically when one builds a trie lexicon from a sorted list of words. We shall come back to this problem below.

2 Generalisation over other free algebras

2.1 Labeled trees

The trees we just treated were just tree skeletons. It is usual to decorate tree nodes with labels in order to represent operator-operand trees, first order terms and formulæ, abstract syntax structures, phrase structure (or dependency structure) of natural languages sentences, etc. We may adapt our zippers easily to these various situations, and even accommodate notions of arities and sorts. But if we wish to enforce too strong invariants on some typing discipline during editing, this is usually counter-productive, since it complicates operations, makes them nonlocal, and forbids editing strategies that go through some ill-typed temporary structure. A typical instance of this problem is when you want to rename some identifier in a program, since either the new or the old name (or most likely both) will have occurrences out of scope during the renaming.

We may also treat λ-calculus expressions, using named variables, or coding them as de Bruijn's indices. Böhm trees, which are co-inductive structures (the corresponding inductive structures being normal λ-terms, represented as layers of head-normal forms) may be also described by zippers. Typed λ-terms will be accommodated either as invariants on untyped terms (à la Curry), or as explicit (raw) typed terms (à la Church). In both cases type-checking is an operation external to editing. Thus we get Automath structures [3], natural deduction proof trees, sequent calculi graphs, and linear logic proof nets representations. In all these cases our structures are acyclic.

We may also label arcs. For instance, we get lexicon structures such as *tries*, where arcs are decorated with letters from the alphabet, and nodes are decorated with booleans (meaning acceptance).

Thus if we consider tries (lexical trees) defined as:

```
type letter = int
and word = list letter;

type trie = [ Trie of (bool * arcs) ]
and arcs = list (letter * trie);
```

then the corresponding trie zippers and focused tries are:

```
type trie_zipper =
  [ Top
  | Zip of (bool * arcs * letter * trie_zipper * arcs)
  ]
and focused_trie = (trie_zipper * trie);
```

and all our algorithms above adapt easily to this case. Tries are good for representing sparse lexicons. Remark that a trie may be considered as the representation of an acyclic deterministic automaton state graph, the boolean marking the corresponding state as accepting or not, the top node representing the initial state, and arcs representing the transitions. This justifies our terminology of arcs in the tree data type, instead of the more usual forest one.

This technology has been used extensively in the Zen Computational Linguistics toolkit [11]. We shall come back to tries in the second part of the paper, after discuting sharing.

2.2 Binary and ternary trees

The zipper operations we surveyed so far were not strictly local in the sense of being constant time primitives, since the `up` operation has to wind up all the elder siblings. Binary trees do not have this drawback, and for them the zipper notion may be reduced to its bare essence.

```
type tree2 = [ Leaf2 | Node2 of (tree2 * tree2) ];

type zipper2 = [ Top
               | Left  of (zipper2 * tree2)
               | Right of (tree2 * zipper2)
               ];
```

as originally given in [9]. In the presentation of [7], these types are named *Bush* and *Context_Bush* respectively.

An alternative presentation would be to use a “siblings stack” style here, as follows.

```
type sum2 = [ Proj21 of tree2 | Proj22 of tree2 ]
and context2 = list sum2
and focus2 = (context2 * tree2);
```

We see that here zippers of binary trees, presented with type context2, are stacks of successive siblings, labeled with their polarity. When we go left in the binary tree, we stack its right sibling with label `Proj21`, and if we go right, we stack its left sibling with label `Proj22`. A focused binary tree is a pair $(context, tree)$ as before. Here are a few operations on the type `(context2 * tree2)`:

```
value left (c,t) = match c with
  [ [ Proj22(s) :: z ] -> ([ Proj21(t) :: z ],s)
  | _ -> raise (Failure "left of top")
  ]
and up (c,t) = match c with
  [ [] -> raise (Failure "up of top")
  | [ Proj21(s) :: z ] -> (z,Node2(t,s))
  | [ Proj22(s) :: z ] -> (z,Node2(s,t))
  ]
and first (c,t) = match t with
  [ Leaf2 -> raise (Failure "first of leaf")
  | Node2(l,r) -> ([Proj21(r)::c],l)
  ];
```

All these operations are constant time, and involve just pointer swapping. They are reminiscent of pointer reversal algorithms, but expressed here in a disciplined manner with applicative well-typed structures.

These operations extend in a straightforward manner to ternary trees:

```
type tree3 = [ Leaf3 | Node3 of (tree3 * tree3 * tree3) ];

type sum3 = [ Proj31 of (tree3 * tree3)
            | Proj32 of (tree3 * tree3)
            | Proj33 of (tree3 * tree3)
            ]
and context3 = list sum3
and focus3 = (context3 * tree3);
```

Such ternary trees may be combined with tries to yield ternary search trees as described by Bentley and Sedgewick [2]. Such trees are optimal to represent lexicons in a well-balanced way. However, in practice the improvement over standard tries is marginal, as discussed in [11].

2.3 Mixed structures

We may further define mixed structures, with trees having zippers as subtructures. This is useful if one keeps several editing structures simultaneously. These mixed structures are adequate to represent mathematical libraries, in proof editors such as Automath or Coq. The zipper substructures would be used for the mathematical theories, with natural sharing of common developments.

Many variations are possible, motivated by applications to structure editing. For instance, it is easy to abbreviate for the user a large context by printing a partial representation of the zipper with holophrasts. This is straightforward by bounding its exploration with some size limit, which is decremented both vertically and horizontally by cost parameters. As one of the referees suggested, one may want to actually reflect this computation by a change of representation of the zipper with appropriate "chunking", with holophrasts represented concertely as mouse-sensitive call-back buttons, for zooming effects. Some experiments are being conducted by the author and a student on a prototype XML editor designed along a client-server architecture, where distributed communication consists in zipper operations byte code. But it is too early to report on this prototype, whose detailed description is beyond the scope of this paper.

2.4 Links with Linear Logic

A focused tree is analogous to a linear logic box:

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

or, in a multisorted algebra, where non-homogeneous substructures may be edited:

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

The zipper is seen as representing a linear function $Z = A \multimap B$ and the focused tree is a pair $S = Z \otimes A = (A \multimap B) \otimes A$, representing lazily this function and its argument. Applying the function amounts to substituting the substructure in the zipped-up one-hole context.

So by computation we retrieve the term from the zipper-context and the subterm, whereas navigation just changes the focus in the corresponding proof net.

Let us try to capture this intuition rigorously. The ML inductive types are systems of recursive equations of the form $T = A_1 \oplus A_2 \oplus ...A_n$ (for a type with n constructors), with each constructor type itself a product: $A_k = A_k^1 \otimes A_k^2 \otimes ...A_k^{p_k}$ (If the k-th constructor has arity p_k). The zipper type Z_T associated to T is: $Z_T = \mathbf{1} \oplus Z_1 \oplus Z_2 \oplus ...Z_n$ with $Z_k = Z_k^1 \oplus Z_k^2 \oplus ...Z_k^{p_k}$ where $Z_k^i = A_k^1 \otimes ...A_k^{i-1} \otimes Z_T \otimes A_k^{i+1}...A_k^{p_k}$ for $1 \leq i \leq p_k$. A focused tree consists in a pair (z, t) where $z : Z_T$ and either $z = Top$ (corresponding to the component $\mathbf{1}$ of Z_T), in which case $t : T$, or else $z = C_k^i(a_1, ..., a_{k-1}, z', a_{k+1}, ..., a_{k+p_k})$ (where C_k^i is the proper canonical injection from Z_k^i to Z_T,) in which case $t : A_k^i$. We assume of course as inductive hypothesis that (z', t') preserves this typing constraint, with $t' = D_k(a_1, ..., a_{k-1}, t, a_{k+1}, ..., a_{k+p_k})$, for D_k the k-th constructor of T. It is straightforward to check that all operations on focused trees respect the typing constraints.

In this general approach, a focused tree has type $Z_T \otimes U$, where U is a sum type of all subcomponents of type T. A simpler account would be to restrict the contexts to type T, by considering only the subcontexts Z_k^i such that $A_k^i = T$ in which case a focused tree has type $Z_T \otimes T$.

As example of this analysis, we take $T = \mathbf{1} \oplus (T \otimes T)$ which yields $Z_T = \mathbf{1} \oplus Z_1 \oplus Z_2$ with $Z_1 = \mathbf{0}$ and $Z_2 = (Z_T \otimes T) \oplus (T \otimes Z_T)$. We thus get for the zipper type of binary trees $Z = \mathbf{1} \oplus (Z_T \otimes T) \oplus (T \otimes Z_T)$, consistent with the ML type `zipper2` above.

Another interesting example is $list\ T = \mathbf{1} \oplus (T \otimes (list\ T))$. From the point of view of the *list* library, the parameter T is an abstract type, and thus should be treated atomically. We thus consider the simpler account. The corresponding zipper structure will yield stacks of elements of type T: $stack\ T = \mathbf{1} \oplus (T \otimes (stack\ T))$. As expected, we get a type isomorphic to $list\ T$, and the "zipper" operations are confused with the data operations on lists - this explains the confusion between lists and stacks which was our original observation.

In the general case, we want to express the transformation from a mutually inductive family of datatypes $T = F(T)$ to a mutually inductive family of their unary contexts $Z_T = G(Z_T, T)$ as a natural transformation from functor F to functor G. Z_T may be seen as $U \multimap T$, for U an indexing set of the subcomponent types of T. A focused structure is like a product indexed on U, whose first component is a zipper in $U \multimap T$ and the second component is a substructure from U. Although the type system of ML is too weak to express directly this dependency, it is possible for every concrete case to define the type of the specific focused structure as a sum type $((U_1 \multimap T) \otimes U_1) \oplus ...((U_N \multimap T) \otimes U_N)$ giving all the ways to construct a structure from a substructure. It is an interesting programming exercise to write a macro-generator of zipper editors as an ML functor. This has been essentially presented by Hinze, Jeuring and Löh in [7], as a characteristic example of what they call a *polytypic function*.

We remark that such formalisations may be carried out statically in proof assistants for type theories with inductive and dependent types such as Coq (in the Calculus of Inductive Constructions). We may thus expect such methodology to be directly definable as a library in programming languages of the future accommodating dependent types.

In a recent paper [15], Conor McBride linked the transformation from functor F to functor G to formal partial differentiation. We believe that this transformation ought to be explained as the construction of the above sum of linar function spaces, which explains linear contexts as representations of linear functions over inductive types. This link with linear logic is justified by the analogy of our zipper operations with Lafont's interaction combinators [13]. Going up and down the zipper amounts to changing the interaction port of the tree constructors to one of the other ports, and conversely. It is an intriguing fact that this construction corresponds to formal partial differentiation - opening speculations about links between analysis and linear logic. Perhaps the proper view is to consider that data structures are integrals of their creating contexts - actually, they abstract the details of their possible progressive linear constructions by zipper computations.

3 Sharing

An important consideration in the design of symbolic manipulation systems such as proof editors is to represent information in a compact manner, sharing common subexpressions as much as possible. Let us show a uniform way of ensuring such sharing.

Sharing data representation is a very general problem. Sharing identical representations is ultimately the responsibility of the runtime system, which allocates and desallocates data with dynamic memory management processes such as garbage collectors.

But sharing of representations of the same type may also be programmed by bottom-up computation. All that is needed is a memo function building the corresponding map without duplications. Let us show the generic algorithm, as an ML *functor*.

3.1 The Share functor

This functor (that is, parametric module) takes as parameter an algebra with its domain seen here as an abstract type. Here is its public interface declaration:

```
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.

```
type bucket = list Algebra.domain;

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

That is, we create the memory as a hash-table array of a given size and of the right bucket type.

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);
```

Now *share x k*, where k is the key of x, looks in k-th bucket l (meaningful since we assume that the key fits in the size: $0 \leq k < Algebra.size$) and returns y in l such that $y = x$ if it exists, and otherwise returns x memorized in the new k-th bucket $[x :: e]$:. Since *share* is the only operation on buckets, we maintain that such y is unique in its bucket when it exists.

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

Instead of *share* we could have used the name *recall*, since either we recall a previously archived equal element, or else this element is archived for future recall. It is an interesting property of this modular design that sharing and archiving are abstracted as a common notion.

We remark that there is no use of pointer equality in this sharing technology. Note that *search* uses ordinary equality, and thus our technique is completely generic over algebraic types, which enjoy structural equality. Of course the ML implementation takes advantage of pointer equality in order to speed up structural equality, but this is another story at a lower level, and pointer equality may be safely hidden from the user, a sanity measure in view of its interference with garbage collection issues.

3.2 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.

```
(* linear hash-code parameters *)
value hash0 = 1
and 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 (trees,index,span) t =
       let (t0,k) = travel t
       in ([t0::trees],index+1,hash1 k index span)
    in let (arcs0,_,span) = List.fold_left f ([],1,hash0) arcs
       in let key = hash span
          in (lookup (Tree(rev arcs0)) key, key)
  ];
```

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;
```

Despite its simplicity, this algorithm is rather efficient, as the benchmarks of [11, 10] indicate. Using hash tables for sharing with bottom-up traversal is a standard dynamic programming technique, but the usual way is to delegate computation of the hash function to some hash library, using a generic low-level package. This is what happens for instance if one uses the module `hashtbl` from the Ocaml library. Here the `Share` module does *not* compute the keys, which are computed on the client side, avoiding re-exploration of the structures. That is, `Share` is just an associative memory. Furthermore, and more importantly, key computation may take advantage of specific statistical distribution of the application domain.

The traversal function, with a proper parametrization of its structure domain by an integer interpretation providing the hash-code parameters, should be constructed generically from the data type definition. This meta-programming looks like an interesting application of the polytypic functions methodology [7].

3.3 Application to automata minimisation

We recalled earlier the notion of lexical tree or *trie*. Tries may be considered as acyclic finite state automata graphs for accepting the (finite) language they represent. This remark is the basis for many lexicon processing libraries. Membership in the trie may be considered as an interpreter for such an automaton, taking its state graph as its trie argument, and its input tape as its word one. Such automata are not minimal, since while the tree structure naturally shares initial subwords, there is no sharing of accepting paths (common final subwords). But this is precisely what sharing does: shrinking a trie into the corresponding dag yields directly the minimal equivalent automaton. This is easy to show, provided the tries are deterministic (every letter occurs at most once in an arcs list) and non redundant (empty subtries are minimally represented as `Trie(False,[])`). This is shown in [10], where many variations of the idea are discussed.

In particular, it is shown in this paper that this idea generalises to more general finite state machines, possibly cyclic, possibly non-deterministic, possibly two-tapes transducers. The general technique is to use a trie as a deterministic skeleton (i.e. as a spanning tree of its state space), and to decorate its nodes with additional information representing the rest of the structure. For instance, non-determinism is represented by choice points decorations. Backpointers in the structure, necessary to represent cyclic state space, are implemented as virtual addresses. Such virtual adresses may be absolute (i.e. using the word designating the corresponding occurrence in the trie) or relative, using the notion of *differential word*. A differential word is a notation permitting to retrieve a word w from another word w' sharing a common prefix, as follows.

```
type delta = (int * word);
```

We compute the difference between `w` and `w'` as a differential word `(|w1|,w2)` where `w=p.w1` and `w'=p.w2`, with maximal prefix `p`. In ML, we compute `diff w w'`, where:

```
value rec diff = fun
    [ [] -> fun x -> (0,x)
    | [c :: r] as w -> fun
        [ [] -> (length w,[])
        | [c' :: r'] as w' -> if c = c' then diff r r'
                              else (length w,w')
        ]
    ];
```

Now `w'` may be retrieved from `w` and `d=diff w w'` as `w'=patch d w`, with:

```
value patch (n,w2) w =
  let p=truncate n (rev w) in unstack p w2;
```

where `truncate n w` is a list library, truncating the initial prefix of length `n` from a word `w`.

The interest of differential words as decorations for relative addresses is that morphological operations may be represented locally, and that the corresponding automata graphs may be shared optimally, leading to efficient structures for storing flexed forms with decorations representing the transducer mapping a flexed form to its stemming information. We refer the reader to [11, 10] for details. Differential words is also the key to incremental construction of tries from sorted lists of words, since the computation of the difference between two words gives the local move in the zipper, without need to zip-up to the top for each item.

The full power of this automaton technology is yet to be researched, since it yields a natural notion of minimal automaton for new families of finite state machines. The crucial property is that of *lexicon morphism*: when the decoration of a node is a function of the corresponding substructure, then sharing the structure will preserve all the sharing of the lexicon, and the decoration of the corresponding dag will represent the automaton as a decoration of its minimal underlying skeleton. This comes for free with sharing - a clear benefit of our applicative methodology, as opposed to the unwieldy standard representations of automata state spaces by spaghetti dishes of pointer structures.

We conclude this section by remarking that our differential words may be seen as zipper operations byte code: the integer part iterates going up, while the word part tells how to go down, the whole thing being the code for navigating in the structure along the shortest path from one node to the other, through their closest common ancestor. This shows in a nutshell that the two techniques we have exhibited are very complementary.

4 Conclusion

We have shown in this paper two techniques for the efficient manipulation of symbolic information structures: unary contexts as zippers and the sharing functor.

Many other design criteria have to be considered when one contemplates the implementation of a formal manipulation system. For instance, a notion of binding operator, and the various techniques to representing bound variables. Existential variables, in the spirit of logic programming, and the problems of pattern matching, unification and constraints processing, in the presence of higher order variables, possibly dependent products, etc. Then the treatment of definitional equality, in the presence of recursion, with rewrite rules, etc. Then problems of opacity or abstraction, and more generally modularity issues. How to organise a corpus or library of formal developments. Tactics programming, and its versioning. Decision procedures and reflection principes. Automation of proof searching. Parsing and printing a mathematical vernacular. Finally organising cooperative work of a community of users with well understood regression testing. All these issues have been the topic of a lot of research during the last 20 years, but there exists at present no comprehensive survey on the proposed solutions to these problems and their mutual interaction.

Acknowledgements. The author wishes to thank the referees of this paper, who made interesting suggestions leading to an improved presentation of the material.

References

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. "Compilers - Principles, Techniques and Tools." Addison-Wesley, 1986.

[2] Jon L. Bentley and Robert Sedgewick. "Fast Algorithms for Sorting and Searching Strings." Proceedings, 8th Annual ACM-SIAM Symposium on Discrete Algorithms, Jan. 1997.

[3] N.G. de Bruijn. "The mathematical language AUTOMATH, its usage and some of its extensions." Symposium on Automatic Demonstration, IRIA, Versailles, 1968. Printed as Springer-Verlag Lecture Notes in Mathematics **125**, (1970) 29–61.

[4] N.G. de Bruijn. "Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem." Indag. Math. **34,5** (1972), 381–392.

[5] Guy Cousineau and Michel Mauny. "The Functional Approach to Programming." Cambridge University Press, 1998.

[6] Philippe Flajolet, Paola Sipala and Jean-Marc Steyaert. "Analytic Variations on the Common Subexpresssion Problem." Proceedings of 17th ICALP Colloquium, Warwick (1990), LNCS 443, Springer-Verlag, pp. 220–234.

[7] R. Hinze, J. Jeuring and A. Löh. "Type-indexed data types." In Mathematics for Program Construction, Springer-Verlag LNCS 2386 (2002).

[8] G. Huet. "An analysis of Böhm's theorem." In "To C. Böhm: Essays on Lambda-Calculus and Functional Programming", eds. M. Dezani-Ciancaglini, S. Ronchi della Rocca and M. Venturini Zilli. Also Theoretical Computer Science 121 (1993) 145–167.

[9] Gérard Huet. "The Zipper". J. Functional Programming 7,5 (Sept. 1997), pp. 549–554.

[10] G. Huet. "Transducers as Lexicon Morphisms, Phonemic Segmentation by Euphony Analysis, Application to a Sanskrit Tagger." Available as: `http://pauillac.inria.fr/~huet/FREE/tagger.ps`.

[11] G. Huet. The Zen Computational Linguistics Toolkit. ESSLLI 2002 Lectures, Trento, Italy, Aug. 2002. Available as: `http://pauillac.inria.fr/~huet/PUBLIC/esslli.pdf`.

[12] A. K. Joshi, and Y. Schabes. "Tree-adjoining grammars." In A. Salomma and G. Rozenberg, Eds., Handbook of Formal Languages and Automata. Springer, Berlin (1997).

[13] Yves Lafont. "Interaction Combinators." Information and Computation 137,1 (1997) pp. 69–101.

[14] Xavier Leroy et al. "Objective Caml." See: `http://caml.inria.fr/ocaml/index.html`.

[15] Conor McBride. "The Derivative of a Regular Type is its Type of One-Hole Contexts." Available from:`http://www.dur.ac.uk/~dcs1ctm/diff.ps`.

[16] Jean-Michel Muller. "Elementary Functions - Algorithms and Implementation." Birkhäuser, 1997.

[17] Daniel de Rauglaudre. "The Camlp4 preprocessor." See: `http://caml.inria.fr/camlp4/`.

[18] Pierre Weis and Xavier Leroy. "Le langage Caml." 2ème édition, Dunod, Paris, 1999.