Backtracking Iterators

Jean-Christophe Filliâtre
CNRS – LRI
Université Paris Sud 91405 Orsay France
filliatr@lri.fr

Abstract

Iterating over the elements of an abstract collection is usually done in ML using a fold-like higher-order function provided by the data structure. This article discusses a different paradigm of iteration based on purely functional, immutable cursors. Contrary to fold-like iterators, the iteration can be cleanly interrupted at any step. Contrary to imperative cursors (such as those found in C++ and Java libraries) it is possible to backtrack the iterator to a previous step. Several ways to iterate over binary trees are examined and close links with Gérard Huet's *Zipper* are established. Incidentally, we show the well-known two-lists implementation of functional queues arising from a *Zipper*-based breadth-first traversal.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Language Constructs and Features*]: Control structures, Data types and structures

General Terms Algorithms, Design

Keywords Iteration, Backtracking, Persistent Data Structures

1. Introduction

The ML programmer is used to iterate over the elements of an abstract collection using a higher-order function. A data structure implemented as an abstract datatype `t`, representing a collection of elements of a given type `elt`, is naturally equipped with a function[1]

```
fold : (elt → α → α) → t → α → α
```

whose behavior is to build a value of type α starting from an initial value (its third argument) and repeatedly applying a function (its first argument) to all elements of the collection (its second argument) and to the value being built. If we are considering a collection of integers `s`, the type `elt` then being `int`, we can sum up all the elements of `s` as simply as

```
fold (fun x n → n+x) s 0
```

When the function passed to `fold` is used only for its side-effects, we can use a degenerated version of `fold`:

```
iter : (elt → unit) → t → unit
```

This way we can print all the elements of `s` as simply as `iter (fun x → Printf.printf "%d\n" x) s`. Such higher-order functions iterating over the elements of a data structure are called *iterators* and their use is probably the most idiomatic feature of functional programming languages. The seasoned ML programmer uses them widely, appreciates their clarity and conciseness, and does not conceive any nicer way to proceed.

There are however some (uncommon) situations where the use of such iterators is not convenient, or even impossible. If the datatype is not abstract, there is usually a most immediate way to iterate over the elements. But if the datatype is *abstract* and we are only given a higher-order iterator, then we may be in a situation where there is no simple or efficient way to implement the desired algorithm.

The first example is the case of an enumeration that must be stopped before the end is reached. If for instance we try to check whether there exists an element in our collection `s` satisfying a given property `p: int → bool`, one solution could be the following:

```
fold (fun x b → p x || b) s false
```

But this is not efficient since all the elements will be necessarily visited, even if we quickly encounter an element satisfying `p` and despite the non-strictness of the `||` operator (since it is applied here to a value and not to a program expression). One solution here is to use an exception to interrupt the iteration. We can write an efficient version of our search using the predefined exception `Exit`:

```
try
  iter (fun x → if p x then raise Exit) s; false
with Exit →
  true
```

Of course, a data structure implementing a collection is usually providing a function `exists : (elt → bool) → t → bool` which does exactly this kind of search, but we always end up in the situation where the search we want to implement is not provided as a primitive. Even if it is efficient, the use of an exception is not always convenient — when a value has to be returned, one needs to defined a custom exception or to use a reference — and is rarely elegant.

This is where the JAVA or C++ programmer is pushing forward his or her way to operate. In such programming languages, iterators do not appear as higher-order functions but as data structures able to produce the elements of the enumeration *one at a time*. Such iterators are often referred to as *cursors* [5]. In JAVA, for instance, the iteration over the elements of a collection `t` is written as the following idiom:

```
for (Iterator i = t.iterator(); i.hasNext(); )
  ... visit i.next() ...
```

[1] This article is illustrated with source code written in OBJECTIVE CAML [2] (OCAML for short), but could be easily adapted to any other functional programming language.

ML'06 September 16, 2006, Portland, Oregon, USA.

The method `iterator` of the data structure `t` builds a new iterator over the elements of `t` and then the two methods `hasNext` and `next` of this iterator respectively tell if there are still elements to iterate over and which is the currently visited element. It is crucial here to understand that the iterator is a *mutable* data structure: one call to `next` returns the current element *and* moves the iterator to the next element in the iteration *by a side-effect*. In the following, we call this a *cursor* to distinguish it from a higher-order iterator.

In most cases of ML programming, such a cursor would be less convenient to use compared to its higher-order counterpart, and much less elegant due to the hidden side-effects. However, it provides a nice solution to the issue of the premature interruption. If we assume an OCAML cursor `i` over our collection `s`, we can easily check for an element satisfying the property `p`:

```
let rec test () =
  has_next i && (p (next i) || test ())
```

Unfortunately, there is (at least) another situation where neither the higher-order iterators nor the cursors can help: when we need to *come back* to a previous state in the iteration, that is when the iterator is involved in a *backtracking* algorithm. As an example, let us assume that we can to check whether our set of integers `s` contains a subset whose sum is 100. If the set `s` would be given concretely as a list of integers, it would be really easy to write a program performing this test[2]:

```
let rec sum n = function
  | [] → n = 100
  | x :: r → sum (n+x) r || sum n r
in
sum 0 s
```

But if the set `s` is implemented by an abstract datatype that only provides higher-order iterators or cursors, then such a backtracking algorithm (that is to try with the current element involved in the sum and then without it in case of failure) is no more possible.

Fortunately there exists a better solution. It consists in a cursor implemented using a *persistent* data structure[3], that is where the move to the next element does not modify the cursor but rather returns a new one. Let us assume that such an cursor is provided as an abstract data type `enum` equipped with two functions `start` and `step`:

```
type enum
val start : t → enum
val step : enum → elt × enum
```

The `start` function builds a new cursor for the whole collection which is given as argument. One can see it as pointing to the "first" element. The `step` function returns the element pointed to by the cursor together with a *new* cursor pointing to the next element. We assume that `step` is raising the `Exit` exception when the iteration is over[4]. Then we can rewrite the backtracking algorithm above as follows:

```
let rec sum n i =
  try
    let x,i = step i in sum (n+x) i || sum n i
  with Exit →
    n = 100
in
sum 0 (start s)
```

The code is highly similar to the one above, apart from lists being replaced by cursors of type `enum`. We can notice that, when the type `t` is actually the type `int list`, we can define `type enum = int list`, the `start` function as the identity and the `step` function as the destructuring function for the constructor `::`.

In this article, we focus on these cursors implemented using persistent data structures, which we call *persistent iterators* in the following. This is not an original technique and such iterators are part of ML folklore. They are used for instance to implement a total ordering over binary search trees in OCAML and SML standard libraries, which is an instance of the samefringe problem [10, 3]. However, the solutions are often *ad-hoc* and the persistence of the iterator is usually not exploited (it is not really mandatory, but imperative programming is simply not considered). The main contribution of this paper is to show how such *ad-hoc* solutions can be retrieved in a systematic way using Gérard Huet's *Zipper* [8]. Incidentally, we show how the well-known implementation of persistent queues using a pair of lists shows up from the *Zipper*-based persistent iterator for breadth-first tree traversal.

Though this is not the main focus of this paper, persistent iterators can also be implemented as *lazy lists* (with or without memoization). Actually, the *ad-hoc* and *Zipper*-based persistent iterators discussed in the following can be seen as *reified* (or *defunctionalized* [6]) versions of lazy lists-based implementations. Thus we briefly present this alternative solution and discuss the connections with our first-order approach.

This article is organized as follows. Section 2 shows how to implement persistent iterators for several traversals over binary trees. Then Section 3 establishes the connections with the *Zipper*. Section 4 shows the equivalence between persistent iterators and lazy lists. Finally, Section 5 quickly compares the performances of all these implementations. The OCAML source code corresponding to what is described in this article is freely available online at `http://www.lri.fr/~filliatr/pub/enum.ml`.

2. Persistent iterators for binary trees

In the following of this article, we assume that the data structure to be iterated over is a binary tree containing integers on nodes:

```
type t = E | N of t × int × t
```

The generalization to balanced trees — thus containing more information within nodes — or trees containing elements of another type is immediate since only the traversals matter here.

This section describes the implementation of persistent iterators for various traversals of binary trees, with the common following signature:

```
type enum
val start : t → enum
val step : enum → int × enum
```

As indicated in the introduction, the `step` function is assumed to raise the `Exit` exception when the enumeration is over.

2.1 Inorder traversal

We start with inorder traversal, which is the most natural traversal when trees are binary search trees. In inorder traversal, the left subtree is visited first, then the element at the node and finally the right subtree. A higher-order iterator `inorder` is thus written as follows:

[2] It is easy to improve this code, but this is not the point here.

[3] The qualifier "persistent" is to be preferred to "purely functional" or "immutable" that are too restrictive. The precise meaning of "persistent" is indeed "observationally immutable", as explained in Okasaki's book [11], and is thus more general.

[4] We could equivalently use a sum type instead of an exception, giving `step` the type `enum → (elt × enum) option`. Using a custom datatype to avoid the indirection, we indeed retrieve the same performances. However, using an exception simplifies most of the forthcoming code.

```
let rec inorder f = function
  | E → ()
  | N (l, x, r) → inorder f l; f x; inorder f r
```

A persistent iterator corresponding to this traversal can be found in the "literature" (the OCAML and SML standard libraries for instance). Since the iteration must begin with the leftmost element in the tree, we start writing a function going left in the tree and building the list of elements and right subtrees encountered meanwhile. We can define a custom list datatype for this purpose:

```
type enum = End | More of int × t × enum
```

and a `left` function implementing the left descent from a given tree `t` and an enumeration `e` representing the elements to be visited after the ones of `t`:

```
let rec left t e = match t with
  | E → e
  | N (l, x, r) → left l (More (x, r, e))
```

We initialize the enumeration with the "empty list" `End`:

```
let start t = left t End
```

and the `step` function is simply returning the element in front of the list and calling `left` with what was the right subtree of this element in the initial tree:

```
let step = function
  | End → raise Exit
  | More (x, r, e) → x, left r e
```

2.2 Preorder traversal

In preorder traversal, the element at the node is visited first, then the elements of the left subtree and finally the elements of the right subtree:

```
let rec preorder f = function
  | E → ()
  | N (l, x, r) → f x; preorder f l; preorder f r
```

It is exactly a depth-first traversal of the tree and thus the iterator can be simply implemented as a *stack*, that is a list of trees:

```
type enum = t list
```

The iterator is initialized with a one element list containing the initial tree:

```
let start t = [t]
```

The `step` function examines the element in first position in the list and, when it is a node, it returns its value while pushing the right and left subtrees on the stack:

```
let rec step = function
  | [] → raise Exit
  | E :: e → step e
  | N (l, x, r) :: e → x, l :: r :: e
```

We can slightly optimize this code to avoid pushing empty trees on the stack:

```
let start = function E → [] | t → [t]
let step = function
  | [] → raise Exit
  | N (E, x, E) :: e → x, e
  | N (E, x, r) :: e → x, r :: e
  | N (l, x, E) :: e → x, l :: e
  | N (l, x, r) :: e → x, l :: r :: e
  | _ → assert false
```

On this example, we can see that the iterator is nothing else than the reification of the call stack. Incidentally, it illustrates another benefit of persistent iterators: to avoid a *stack overflow*. Even if in the case of balanced binary trees it is unlikely that the depth of a tree can be responsible for a stack overflow, the case of other data structures, such as graphs for instance, can be more problematic for iterators simply written as recursive functions. Of course, it is always possible to make the stack explicit, even in the case of usual higher-order iterators.

2.3 Postorder traversal

In postorder traversal, we visit the element at the node *after* having visited the elements of the two subtrees. Surprisingly, postorder traversal is more difficult to implement than preorder traversal. Of course, we could reuse the idea of making the call stack explicit and pushing elements as well as trees. But it would not be an efficient solution. More subtly, we can reuse ideas from the inorder traversal since the first element to be visited is also the leftmost element in the tree. Thus we reuse the same iterator type and the `left` and `start` functions:

```
type enum = End | More of int × t × enum
let rec left t e = match t with
  | E → e
  | N (l, x, r) → left l (More (x, r, e))
let start t = left t End
```

Only the `step` function needs to be updated. It must now consider the right subtree `r` *before* the element `x`:

```
let rec step = function
  | End →
      raise Exit
  | More (x, E, e) →
      x, e
  | More (x, r, e) →
      step (left r (More (x, E, e)))
```

Pushing the empty tree `E` together with `x` on the last case is not very elegant. We can refine this solution by introducing a custom constructor `More1` to handle this particular case:

```
type enum =
  | End
  | More of t × int × enum
  | More1 of int × enum
let rec left t e = match t with
  | E → e
  | N (l, x, E) → left l (More1 (x, e))
  | N (l, x, r) → left l (More (r, x, e))
let start t = left t End
let rec step = function
  | End → raise Exit
  | More1 (x, e) → x, e
  | More (t, x, e) → step (left t (More1 (x, e)))
```

2.4 Breadth-first traversal

We end this section devoted to binary trees with breadth-first traversal. It is usually implemented using a *queue* containing trees. The whole tree is inserted into an initially empty queue and then, for each tree popped out of the queue, we visit the element at the node and insert the two subtrees into the queue (the left one and then the right one). Writing the usual higher-order iterator using the imperative queues from OCAML standard library is straightforward:

```
let bfs f t =
  let q = Queue.create () in
  Queue.push t q;
  while not (Queue.is_empty q) do
    match Queue.pop q with
```

```
    | E → ()
    | N (l, x, r) →
        f x; Queue.push l q; Queue.push r q
  done
```

To implement the corresponding persistent iterator, we simply need to substitute *persistent queues* to imperative queues. It happens that it is quite easy to implement persistent queues using a pair of lists while keeping good performances [11]. The code is given in appendix as a module Q implementing as abstract datatype α t for persistent queues containing elements of type α. The persistent iterator is then directly implemented as a persistent queue containing trees:

```
type enum = t Q.t
```

The start function builds the queue containing only one element, namely the whole tree:

```
let start t = Q.push t Q.empty
```

and the step function applies the same algorithm as above:

```
let rec step e =
  try match Q.pop e with
    | E, e → step e
    | N (l, x, r), e → x, Q.push r (Q.push l e)
  with Q.Empty →
    raise Exit
```

Note: as we did for the preorder traversal, it is possible to slightly optimize this code by avoiding pushing empty trees in the queue. This remark also applies to the imperative algorithm, of course.

3. Connections with the zipper

In this section, we investigate the connections between the persistent iterators and Gérard Huet's *Zipper* [8, 7]. More precisely, we show how persistent iterators can be discovered in a systematic way using the *Zipper*.

3.1 The zipper

We introduce the *Zipper* for the reader who would not be familiar of this data structure. The *Zipper* is to the purely applicative data structure what the pointer is to a mutable data structure: a way to designate a piece of the structure and to modify it. In the case of a purely applicative data structure, "to modify" of course means building a new value but this does not simplify the issue. Let us assume we are visiting the nodes of a binary tree looking for some node satisfying a given property and, once it has been found, we want to perform a local modification. With an imperative data structure, it is immediate. But with an applicative data structure, we need to maintain the path from the root of the tree to the visited node, to be able to rebuild the corresponding nodes. That is precisely what the *Zipper* does, with the greatest elegance.

Such a path from the root is represented in a bottom-top way, as a list going from the visited node to the root, the direction followed at each step being indicated. The OCAML type for this path is the following:

```
type path =
  | Top
  | Left of path × int × t
  | Right of t × int × path
```

The *Zipper* is then the pair of the subtree which is "pointed to" and of the path to the root:

```
type location = t × path
```

The construction can be generalized to any algebraic datatype, each constructor being duplicated into several variants (in our case, the constructor N is duplicated into Left and Right).

We create a *Zipper* pointing to the root of a tree t by associating it to the empty path:

```
let create t = (t, Top)
```

Then we can build *navigation* functions allowing to move in the tree represented by the *Zipper*. To descend to the left subtree, when there is one, we simply need to extend to path with the Left constructor, which records the value at the node and the right subtree, and then to take the left subtree as the new designated tree:

```
let go_down_left = function
  | E, _ → invalid_arg "go_down_left"
  | N (l, x, r), p → l, Left (p, x, r)
```

Symmetrically, we can define a function to descend to the right subtree:

```
let go_down_right = function
  | E, _ → invalid_arg "go_down_right"
  | N (l, x, r), p → r, Right (l, x, p)
```

Similarly, we can define functions to move from a tree to its left or right sibling, when they exist:

```
let go_left = function
  | _, Top | _, Left _ → invalid_arg "go_left"
  | r, Right (l, x, p) → l, Left (p, x, r)

let go_right = function
  | _, Top | _, Right _ → invalid_arg "go_right"
  | l, Left (p, x, r) → r, Right (l, x, p)
```

Finally, we can define a function to move up in the tree:

```
let go_up = function
  | _, Top → invalid_arg "go_up"
  | l, Left (p, x, r) | r, Right (l, x, p) →
      N (l, x, r), p
```

Iterating this function until we reach the empty path Top is a way to retrieve the whole tree represented by the *Zipper*. The local modification, which was the motivation for the *Zipper*, is trivially implemented as a replacement of the designated subtree with a new one:

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

We note that all these operations are implemented in constant time and space.

3.2 Persistent iterators derived from the zipper

We now show how the *Zipper* can be used to retrieve the persistent iterators over binary trees introduced in Section 2.

3.2.1 Inorder traversal

Let us start with the inorder traversal. The persistent iterator is directly represented by a *Zipper* and the start function sets the *Zipper* to the root of the tree:

```
type enum = location
let start t = (t, Top)
```

The step function is then implemented by combining the navigation functions provided by the *Zipper* and the local modification function to get rid of the visited elements one at a time. If the whole tree is empty, then the iteration is over:

```
let rec step = function
  | E, Top → raise Exit
```

If the designated subtree is a node, then we must keep descending to the left, using the *Zipper* primitive `go_down_left`, and call `step` recursively. If we expand `go_down_left`, we get:

```
| N (l, x, r), p → step (l, Left (p, x, r))
```

Finally, when we reach the empty tree `E`, the element `x` right above is the one to visit and we can replace the designated node by its right sibling. This is achieved by a combination of the primitives `go_up` and `change`. Once expanded, we get the following code:

```
| E, Left (p, x, r) → x, (r, p)
```

To sum up, the `step` function is only three lines long:

```
let rec step = function
  | E, Top → raise Exit
  | E, Left (p, x, r) → x, (r, p)
  | N (l, x, r), p → step (l, Left (p, x, r))
```

We notice that the *Zipper* constructor `Right` has not been used and thus can be eliminated:

```
type path = Top | Left of path × int × t
type enum = t × path
```

Consequently, we obtain *exactly* the datatype introduced in Section 2, that is a list of pairs composed of elements and their associated right subtrees. The `left` function from the initial solution has disappeared: it is now encoded directly by the `step` function. The behavior is slightly different, though: some calls to `left` in the initial solution are now suspended in the first component of the *Zipper* and will only be performed on the next call to `step`. If we stop an iteration on a node with no left subtree and a huge right subtree (to the left) then we save the descent to the left in this huge tree. As a consequence, the solution inspired by the *Zipper* is slightly more efficient.

3.2.2 Preorder traversal

We keep implementing the persistent iterator directly as the *Zipper* and the `start` function is unchanged, as is the termination case of the iteration:

```
let rec step = function
  | E, Top → raise Exit
```

If the visited subtree is a node, then we return its element and we move the *Zipper* to the left subtree (go_down_left):

```
| N (l, x, r), p → x, (l, Left (p, x, r))
```

Finally, when the iteration reaches the leftmost element, we immediately jump to the right subtree since the element at the node has already been visited:

```
| E, Left (p, _, r) → step (r, p)
```

Putting all together, we get the following code:

```
let rec step = function
  | E, Top → raise Exit
  | E, Left (p, _, r) → step (r, p)
  | N (l, x, r), p → x, (l, Left (p, x, r))
```

Once again we note that the constructor `Right` is useless, and so is the element stored in the constructor `Left`. Thus we can simplify the definition of the iterator into

```
type path = Top | Left of path × t
type enum = t × path
```

We find again *exactly* the same datatype as in the initial solution, namely a list of trees (with a particular case for the first element, represented as a pair). As far as efficiency is concerned, this solution inspired by the *Zipper* is intermediate between the two solutions proposed in Section 2, since it avoids pushing some empty trees on the stack, but not all of them.

3.2.3 Postorder traversal

The persistent iterator datatype, the `start` function and the termination case for `step` are still unchanged:

```
let start t = (t, Top)
let rec step = function
  | E, Top → raise Exit
```

If the tree pointed to is a node, we need to descend into its left subtree:

```
| N (l, x, r), p → step (l, Left (p, x, r))
```

If the iteration is done with a left subtree, it must now consider its right sibling:

```
| E, Left (p, x, r) → step (r, Right (E, x, p))
```

Finally, if the iteration reaches the rightmost element, it simply needs to return this element and to suppress the corresponding node:

```
| E, Right (_, x, p) → x, (E, p)
```

We get the following code for `step`:

```
let rec step = function
  | E, Top → raise Exit
  | E, Left (p, x, r) → step (r, Right (E, x, p))
  | E, Right (_, x, p) → x, (E, p)
  | N (l, x, r), p → step (l, Left (p, x, r))
```

Here, both *Zipper*'s constructors `Left` and `Right` are used but we notice that the first argument of `Right` is not used. Thus we can slightly simplify the definition of the persistent iterator into

```
type path =
  | Top
  | Left of path × int × t
  | Right of int × path
type enum = t × path
```

Once again, we find out a datatype isomorphic to the one introduced in Section 2 (`Left` corresponding to `More` and `Right` to `More1`).

3.2.4 Breadth-first traversal

The case of breadth-first traversal is more complex. Indeed, using only the navigation primitives provided by the *Zipper* to move from one node to the next node in the breadth-first traversal is quite difficult: one needs to come back to an upper node in the tree and then to move down following another branch, in a way that depends on the *global* structure of the tree.

As usual with breadth-first traversals, we need to generalize the problem to *forests* (see for instance [12]), that is to lists of trees. Indeed, it is therefore possible to represent the list of all the subtrees of a same level and then to move from one node to its right sibling in this forest.

There happens to be a *Zipper* for variadic arity trees and thus for forests. In the original paper introducing the *Zipper* [8] the case of variadic arity trees is even considered before the particular case of binary trees. The *Zipper* is defined as follows:

```
type path = Top | Node of t list × path × t list
type location = t × path
```

The three arguments of the `Node` constructor represent a position within a forest, the first list containing the trees on the left *in reverse* *order* and the second list the trees on the right. The navigation primitives that are of interest here are the following:

```
let go_left = function
  | t, Node (l :: ll, p, r) →
     l, Node (ll, p, t :: r)
  | _ → invalid_arg "go_left"

let go_right = function
  | t, Node (l, p, r :: rr) →
     r, Node (t :: l, p, rr)
  | _ → invalid_arg "go_right"
```

As with the previous traversals, the persistent iterator is directly represented by the *Zipper* and the `start` function sets the *Zipper* on the root of the tree:

```
type enum = location
let start t = t, Node ([], Top, [])
```

As previously, the `step` function is implemented using the navigation primitives and removing the elements as soon as they are visited. The case of the empty forest terminates the iteration:

```
let rec step = function
  | E, Node ([], _, []) → raise Exit
```

If the designated tree is a node, we return the corresponding element and we replace this tree by its left and right subtrees, pushed onto the left list. In order to avoid considering too many particular cases, we replace the designated tree by an empty tree:

```
  | N (l, x, r), Node (ll, p, rr) →
      x, (E, Node (r :: l :: ll, p, rr))
```

If the designated tree is precisely an empty tree, then we move right into the forest:

```
  | E, Node (ll, p, r :: rr) →
      step (r, Node (ll, p, rr))
```

Finally, if it is no more possible to move right, we need to come back to the leftmost position in the forest (to move to the next level). This amounts to applying the `go_left` function repeatedly as much as possible, which results in the reversing of the left list into the right list (in the efficient way, that is using an accumulator which is here the right list). Thus we can use `List.rev` directly:

```
  | E, Node (ll, p, []) →
      step (E, Node ([], p, List.rev ll))
```

Putting all together, we get the following code:

```
let rec step = function
  | E, Node ([], p, []) → raise Exit
  | E, Node (ll, p, []) →
      step (E, Node ([], p, List.rev ll))
  | E, Node (ll, p, r :: rr) →
      step (r, Node (ll, p, rr))
  | N (l, x, r), Node (ll, p, rr) →
      x, (E, Node (r :: l :: ll, p, rr))
  | _, Top → assert false
```

We immediately notice that the *Zipper* is always of the kind `Node(_,Top,_)`. Thus we can suppress the `Top` and `Node` constructors and represent here the *Zipper* by a pair of lists. We can also put the designated subtree in head position of the right list, which gives the final code below:

```
type enum = t list × t list
let start t = [], [t]
let rec step = function
  | [], [] → raise Exit
  | ll, [] → step ([], List.rev ll)
  | ll, E :: rr → step (ll, rr)
  | ll, N (l, x, r) :: rr → x, (r :: l :: ll, rr)
```

The interpretation of the two lists composing the iterator is quite simple: one list stores the nodes still to be traversed on the current level, while the other one collects the nodes of the next level, in reverse order. Reversal occurs precisely when the iterator moves from one level to the next one.

It appears that this is *exactly* the solution given in Section 2.4, except that the code for persistent queues using pairs of lists (see the appendix) is here *inlined* in the `step` function. Incidentally, we have retrieved the efficient coding of persistent queues while using the *Zipper* to perform a breadth-first traversal.

4. Lazy lists and continuation passing style

In this section, we briefly explore an alternative solution where persistent iterators are implemented as *lazy lists*. Memoization set apart, a lazy list is simply a function returning its first element together with the remaining elements as another lazy list. Thus a persistent iterator is represented directly as a function giving the next element together with the new iterator[5]:

```
type enum = unit → int × enum
```

Therefore the `step` function is simply an application of this function:

```
let step k = k ()
```

As a consequence, all the algorithmic complexity is moved to the `start` function, which must build a single closure containing all the forthcoming computation. At this point, it is useful to switch to *continuation passing style* (CPS) and to define a more general function, called `run` here, which takes a continuation as argument and to initiate the computation with the "empty" continuation that raises the `Exit` exception to signal the end of the iteration:

```
let run t k = ...
let start t = run t (fun () → raise Exit)
```

Defining the `run` function for inorder, preorder and postorder traversals is rather straightforward:

Inorder traversal

```
let rec run t k = match t with
  | E → k
  | N (l, x, r) →
      run l (fun () → (x, run r k))
```

Preorder traversal

```
let rec run t k = match t with
  | E → k
  | N (l, x, r) →
      (fun () → (x, run l (run r k)))
```

Postorder traversal

```
let rec run t k = match t with
  | E → k
  | N (l, x, r) →
      run l (run r (fun () → (x, k)))
```

[5] Such a type definition requires to set the `-rectypes` option of the OCAML compiler. We could do without, using a boxed type such as

```
type enum = F of (unit -> int * enum)
```

but the resulting boxing and unboxing imply a severe loss of performance, at least in OCAML (up to 60% slower code in our tests).

traversal	implementation	random	left	right	full
inorder	Section 2.1	**1.07**	**0.86**	**0.11**	**0.25**
	zipper	1.14	1.12	**0.11**	0.27
	lazy list	1.28	1.38	0.12	0.29
preorder	Section 2.2	1.01	0.76	0.10	0.26
	— variant	**0.91**	**0.08**	**0.07**	**0.21**
	zipper	0.99	0.99	0.11	0.27
	lazy list	1.51	0.14	0.16	0.41
postorder	Section 2.3	1.26	0.86	1.04	**0.28**
	— variant	**1.20**	**0.69**	**0.87**	0.29
	zipper	1.42	1.08	1.06	0.35
	lazy list	1.63	1.44	1.21	0.43
bfs	Section 2.4	14.57	0.44	0.47	4.62
	— variant	**9.26**	0.24	0.24	**2.05**
	zipper	15.38	**0.18**	**0.17**	3.65

Figure 1. Compared performances of the various implementations

This solution is somewhat systematic. Unfortunately, it is less efficient than the previous approaches, mostly because closures and their applications are using much more time and space than custom data structures (some benchmarks are presented in the next section). To improve the efficiency of this code, we could *defunctionalize* it i.e. replace the closures by some custom data type [6]. Then we would find out implementations roughly similar to approaches of sections 2 and 3. Thus our persistent iterators and lazy lists are actually equivalent.

There is even a systematic way to turn a fold-like (or iter-like) iterator into lazy lists using delimited continuations [3, 9]. Thus it is always possible to get persistent iterators for any data structure providing a traditional higher-order iterator, even when the data structure is implemented as an abstract datatype.

5. Performance

In this section we quickly compare the efficiency of the various solutions proposed in this article. The Figure 1 gathers the timings performed for the various kinds of traversals and the various implementations. Each implementation is tested against a large set of trees of various sizes (between 0 and 100000 elements) and we measure the total time spent in the full traversals of these trees. Four different shapes of trees are considered: random trees, left-linear trees, right-linear trees and full trees. The iterators introduced in Section 2 are named "Section 2.1" to "Section 2.4" and "variant" refers to the optimizations (where we do not push empty trees); "zipper" refers to the solutions introduced in Section 3; finally, "lazy list" refers to the solutions introduced in the previous section. The code was compiled with the OCAML native-code compiler (`ocamlopt`, without any option apart from `-rectypes`) on a Pentium 4 processor running under Linux. The timings are given in seconds and correspond to CPU time obtained using the UNIX `times` system call.

We notice that the persistent iterators described in Section 2 are slightly more efficient than the versions derived from the *Zipper*. This is due to a slightly more immediate data structure (there is no pair at the top of the data structure), but the difference is not significant. In the case of the breadth-first traversal, we notice that the *Zipper* solution is sometimes more efficient, but this is due to the inlining of the persistent queues implementation. Last, we notice that the lazy lists are always less efficient[6]. There are at least two possible explanations. First, a closure is usually larger than a *ad-hoc* data structure, thus resulting in a slightly more expensive allocation and initialization. Second, modern processors can deal more efficiently with pattern matching (using branch prediction) than with unknown function calls.

We also performed a similar comparison for the memory use, measuring the evolution of the size of the persistent iterator. Results are very similar to those of time performances: solutions from Section 2 and those inspired by the *Zipper* use similar amounts of memory, and the lazy lists solution is using much more memory (but within a constant factor).

The results of these benchmarks can be reproduced using the source code available online.

[6] apart from the very special case of a preorder traversal of a left-linear tree, which is then equivalent to the usual traversal of a list and thus quasi optimal.

6. Conclusion

In this article, we have considered an alternative to the traditional higher-order iterators that can be found in ML, as step-by-step iterators based on persistent data structures. These *persistent iterators* allow the premature interruption of an iteration and, even better, the resumption on a previous state of the iteration, which is useful when the iterator is involved in a backtracking algorithm.

Such iterators are known from some programmers, though their persistence is usually not exploited. They are however not widely spread and deserve more advertisement. We have already done a step in that direction in the OCAMLGRAPH [1, 4] library: some persistent iterators are provided for depth-first and breadth-first graph traversals and are used in a backtracking-based graph coloring algorithm. Similarly, one could provide persistent iterators for other usual data structures such as hash tables, queues, stacks, etc.

This article also pointed out the close connections with Gérard Huet's *Zipper*: using the navigation primitives provided by the *Zipper*, we could easily find out persistent iterators implementations for various binary trees traversals. As a bonus, we rediscovered the two-lists efficient implementation of persistent queues while studying a *Zipper*-based breadth-first traversal.

Finally, we think that a more systematic study of persistent iterators would also help implementing *imperative* cursors (as used in object-oriented programming libraries), which is notoriously difficult but has received little consideration so far [13].

A. Appendix: Persistent queues

We give here a simple but efficient implementation of persistent queues [11]. The idea is to represent a queue as a pair of lists, one to insert the new elements (in head of the list) and the other to extract the elements (also in head). We may have eventually to reverse the first list whenever the second one becomes empty, but the *amortized* complexity of `push` and `pop` is still $O(1)$.

```
module Q : sig
  type α t
  exception Empty
  val empty : α t
  val is_empty : α t → bool
  val push : α → α t → α t
  val pop : α t → α × α t
end = struct
  type α t = α list × α list
  exception Empty
  let empty = [], []
  let is_empty = function [], [] → true | _ → false
  let push x (i,o) = (x :: i, o)
  let pop = function
    | i, y :: o → y, (i,o)
    | [], [] → raise Empty
```

```
    | i, [] → match List.rev i with
        | x :: o → x, ([], o)
        | [] → assert false
end
```

Acknowledgments

I am very grateful to Oleg Kiselyov for his highly detailed comments on the first version of this paper, and to the anonymous referees for their suggestions. I am also grateful to Sylvain Conchon, Benjamin Monate and Julien Signoles for informal discussions related to this paper.

References

[1] Ocamlgraph, a graph library for Objective Caml. `http://www.lri.fr/~filliatr/ocamlgraph/`.

[2] The Objective Caml language. `http://caml.inria.fr/`.

[3] Dariusz Biernacki, Olivier Danvy, and Chung chieh Shan. On the Static and Dynamic Extents of Delimited Continuations. Technical Report RS-05-36, BRICS, 2005. To appear in the journal Science of Computer Programming.

[4] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a Generic Graph Library using ML Functors. Submitted for publication. `http://www.lri.fr/~filliatr/ocamlgraph/main.ps`, May 2006.

[5] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[6] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at Work. In *Søndergaard, editor, 3rd International Conference on Principles and Practice of Declarative Programming (PPDP '01)*, June 2001.

[7] Ralf Hinze and Johan Jeuring. Functional Pearl: Weaving a Web. *Journal of Functional Programming*, 11(6):681–689, November 2001.

[8] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, Septembre 1997.

[9] Oleg Kiselyov. From enumerators to cursors: turning the left fold inside out, 2004. At `http://okmij.org/ftp/Haskell/fold-stream.lhs`.

[10] John McCarthy. Another samefringe. *SIGART Newsletter*, 61, February 1977.

[11] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[12] Chris Okasaki. Breadth-First Numbering: Lessons from a Small Exercise in Algorithm Design. In *International Conference on Functional Programming (ICFP)*, Montreal, Canada, 2000.

[13] Richard Rasala. A Model C++ Tree Iterator Class For Binary Search Trees. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education SIGCSE*, 1997.