Backtracking Iterators

Jean-Christophe Filliâtre
CNRS – Université Paris Sud

ML Workshop, Portland, Oregon

September 16th, 2006

ML Iterators

abstract datatype `t` for a **collection** of elements of type `elt`

it is usual to provide **iterators** such as

`fold : (elt` $\rightarrow \alpha \rightarrow \alpha$`)` $\rightarrow$ `t` $\rightarrow \alpha \rightarrow \alpha$

and

`iter : (elt` $\rightarrow$ `unit)` $\rightarrow$ `t` $\rightarrow$ `unit`

massively used: one occurrence each 40 lines of code

ML Iterators

abstract datatype `t` for a **collection** of elements of type `elt`

it is usual to provide **iterators** such as

$$\texttt{fold : (elt} \rightarrow \alpha \rightarrow \alpha\texttt{)} \rightarrow \texttt{t} \rightarrow \alpha \rightarrow \alpha$$

and

$$\texttt{iter : (elt} \rightarrow \texttt{unit)} \rightarrow \texttt{t} \rightarrow \texttt{unit}$$

massively used: one occurrence each 40 lines of code

ML Iterators

abstract datatype `t` for a **collection** of elements of type `elt`

it is usual to provide **iterators** such as

fold : (elt $\rightarrow \alpha \rightarrow \alpha) \rightarrow$ t $\rightarrow \alpha \rightarrow \alpha$

and

iter : (elt $\rightarrow$ unit) $\rightarrow$ t $\rightarrow$ unit

massively used: one occurrence each 40 lines of code

Drawbacks

in some situations, the higher-order iterators are not convenient

1. premature **interruption**
   e.g. is there an element in the collection satisfying `p:elt→bool`?

2. **simultaneous** traversal of several collections
   e.g. the same-fringe problem (do two binary trees have the same leaves when read from left to right?)

3. **backtracking**
   e.g. greedy graph-coloring algorithm based on a given iterator to visit the graph nodes (DFS, BFS, etc.)

Drawbacks

in some situations, the higher-order iterators are not convenient

1. premature **interruption**

   e.g. is there an element in the collection satisfying `p:elt→bool`?

2. **simultaneous** traversal of several collections

   e.g. the same-fringe problem (do two binary trees have the same leaves when read from left to right?)

3. **backtracking**

   e.g. greedy graph-coloring algorithm based on a given iterator to visit the graph nodes (DFS, BFS, etc.)

Drawbacks

in some situations, the higher-order iterators are not convenient

1. premature **interruption**
   e.g. is there an element in the collection satisfying `p:elt→bool`?

2. **simultaneous** traversal of several collections
   e.g. the same-fringe problem (do two binary trees have the same leaves when read from left to right?)

3. **backtracking**
   e.g. greedy graph-coloring algorithm based on a given iterator to visit the graph nodes (DFS, BFS, etc.)

Cursors in Object-Oriented Programming

an iterator is an object of a class such as

```
class Iterator {
  boolean hasNext();
  Object next();
}
```

and is used with the following pattern

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

nicely addresses drawbacks 1 (interruption) and 2 (simultaneous traversals) but inelegant (hidden side effect) and does not allow backtracking

Cursors in Object-Oriented Programming

an iterator is an object of a class such as

```
class Iterator {
  boolean hasNext();
  Object next();
}
```

and is used with the following pattern

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

nicely addresses drawbacks 1 (interruption) and 2 (simultaneous traversals)
but inelegant (hidden side effect) and does not allow backtracking

Cursors in Object-Oriented Programming

an iterator is an object of a class such as

```
class Iterator {
  boolean hasNext();
  Object next();
}
```

and is used with the following pattern

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

nicely addresses drawbacks 1 (interruption) and 2 (simultaneous traversals)
but inelegant (hidden side effect) and does not allow backtracking

Persistent Iterators

a different paradigm of iteration: a **persistent iterator**

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

nice solution to our three issues

already exists in the ML folklore

Persistent Iterators

a different paradigm of iteration: a **persistent iterator**

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

nice solution to our three issues

already exists in the ML folklore

Persistent Iterators

a different paradigm of iteration: a **persistent iterator**

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

nice solution to our three issues

already exists in the ML folklore

Binary Tree Comparison

total ordering of binary search trees in the Ocaml or SML standard libraries

```
type t = E | N of t × int × t

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

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

Binary Tree Comparison

total ordering of binary search trees in the Ocaml or SML standard libraries

```
type t = E | N of t × int × t

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

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

Binary Tree Comparison

total ordering of binary search trees in the Ocaml or SML standard libraries

```
type t = E | N of t × int × t

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

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

Binary Tree Comparison

total ordering of binary search trees in the Ocaml or SML standard libraries

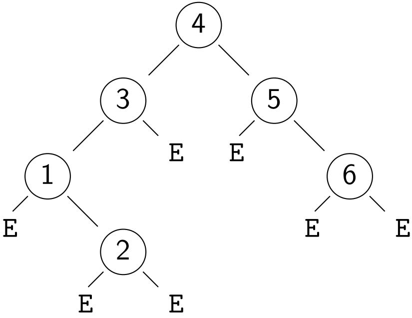
```
type t = E | N of t × int × t

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

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

Binary Tree Comparison

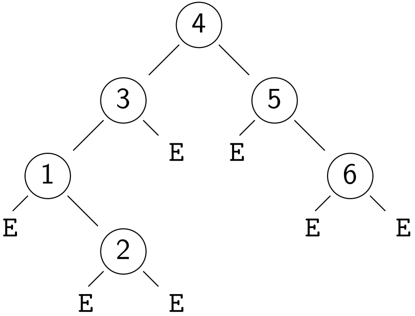
total ordering of binary search trees in the Ocaml or SML standard libraries

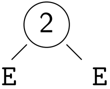
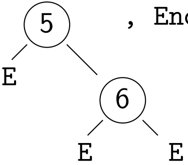
```
type t = E | N of t × int × t

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

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

Example

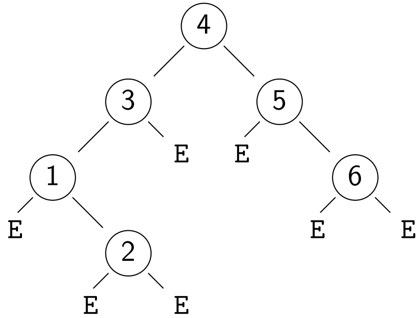


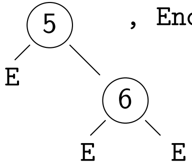
Example



More(1,  , More(3, E, More(4,  , End)))

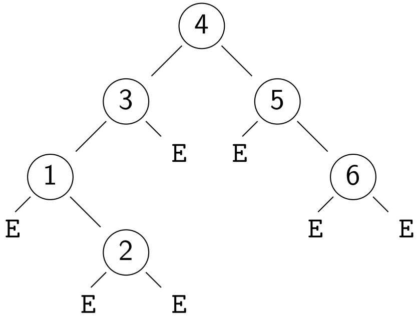
Example

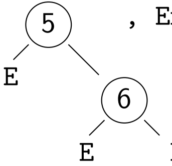


1 More(2, E, More(3, E, More(4, 

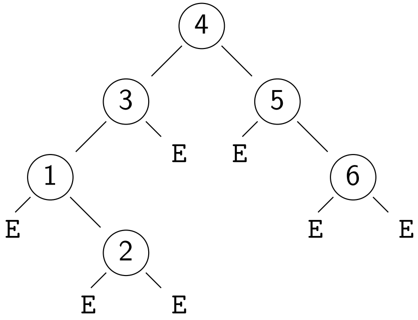
, End)))

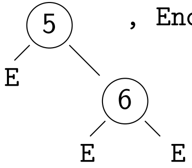
Example



1 2 `More(3, E, More(4,`  `, End))`

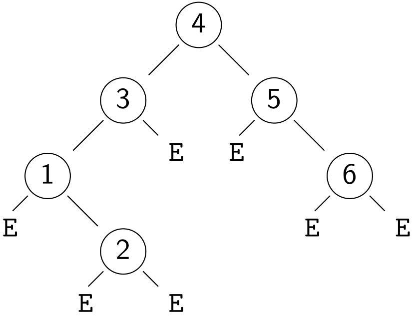
Example

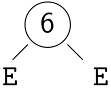


1 2 3 More(4, 

, End)

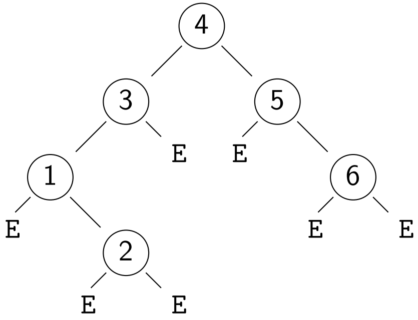
Example



1 2 3 4 More(5, 

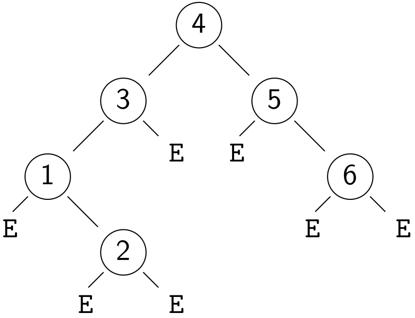
, End)

Example



1 2 3 4 5 More(6, E, End)

Example



1 2 3 4 5 6 End

A Systematic Way?

we can build similar **ad-hoc** persistent iterators for preorder, postorder or breadth-first traversals (cf the proceedings)

but is there a **systematic** way to build persistent iterators?

at least one: **lazy lists**

a lazy list is a **function** returning its first element and the lazy list of the remaining elements, i.e.

```
type enum = unit → int × enum
```

hence `step` is immediate

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

A Systematic Way?

we can build similar **ad-hoc** persistent iterators for preorder, postorder or breadth-first traversals (cf the proceedings)

but is there a **systematic** way to build persistent iterators?

at least one: **lazy lists**

a lazy list is a **function** returning its first element and the lazy list of the remaining elements, i.e.

```
type enum = unit → int × enum
```

hence `step` is immediate

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

Lazy List Example: Inorder Traversal

it is convenient to switch to Continuation Passing Style

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

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

if efficiency really matters, we can even **defunctionalize** the lazy list

then we get solutions roughly similar to the ad-hoc data structures

Lazy List Example: Inorder Traversal

it is convenient to switch to Continuation Passing Style

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

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

if efficiency really matters, we can even **defunctionalize** the lazy list

then we get solutions roughly similar to the ad-hoc data structures

Lazy List Example: Inorder Traversal

it is convenient to switch to Continuation Passing Style

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

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

if efficiency really matters, we can even **defunctionalize** the lazy list

then we get solutions roughly similar to the ad-hoc data structures

Another Approach

Gérard Huet. *The Zipper*. Journal of Functional Programming, 1997

the **zipper** is to the **applicative** structure
what the **pointer** is to the **imperative** structure

it is a mean to move within an applicative structure
and to make local **modifications**

Another Approach

Gérard Huet. *The Zipper*. Journal of Functional Programming, 1997

the **zipper** is to the **applicative** structure
what the **pointer** is to the **imperative** structure

it is a mean to move within an applicative structure
and to make local **modifications**

Another Approach

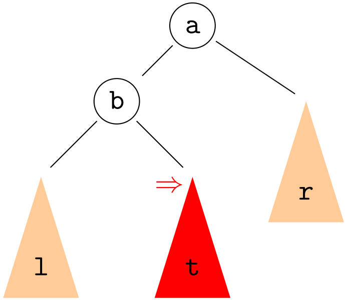
Gérard Huet. *The Zipper*. Journal of Functional Programming, 1997

the **zipper** is to the **applicative** structure
what the **pointer** is to the **imperative** structure

it is a mean to move within an applicative structure
and to make local **modifications**

Zipper for Binary Trees

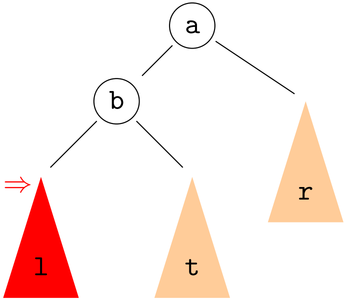
```
type path = Top | Left of path × int × t | Right of t × int × path
type location = t × path
```



l, Left(Left(Top, a, r), b, t) ⇔ t, Right(l, b, Left(Top, a, r))

Zipper for Binary Trees

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



`l, Left(Left(Top, a, r), b, t)` ⇔ `t, Right(l, b, Left(Top, a, r))`

Zipper for Binary Trees

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

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

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

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

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

Zipper for Binary Trees

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

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

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

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

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

Zipper for Binary Trees

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

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

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

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

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

Zipper for Binary Trees

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

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

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

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

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

Zipper for Binary Trees

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

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

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

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

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

Inorder Traversal using the Zipper

persistent iterator = zipper

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

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

constructor Right not even used ⇒

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

type **isomorphic** to

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

Inorder Traversal using the Zipper

persistent iterator = zipper

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

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

constructor Right not even used ⇒

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

type **isomorphic** to

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

Inorder Traversal using the Zipper

persistent iterator = zipper

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

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

constructor Right not even used ⇒

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

type **isomorphic** to

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

Inorder Traversal using the Zipper

persistent iterator = zipper

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

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

constructor Right not even used ⇒

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

type **isomorphic** to

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

Inorder Traversal using the Zipper

persistent iterator = zipper

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

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

constructor Right not even used ⇒

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

type **isomorphic** to

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

Inorder Traversal using the Zipper

persistent iterator = zipper

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

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

constructor Right not even used ⇒

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

type **isomorphic** to

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

Inorder Traversal using the Zipper

persistent iterator = zipper

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

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

constructor Right not even used ⇒

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

type **isomorphic** to

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

Inorder Traversal using the Zipper

persistent iterator = zipper

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

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

constructor `Right` not even used ⇒

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

type **isomorphic** to

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

Similarly...

for preorder and postorder traversals, we retrieve solutions isomorphic to the ad-hoc ones using the Zipper (cd the proceedings)

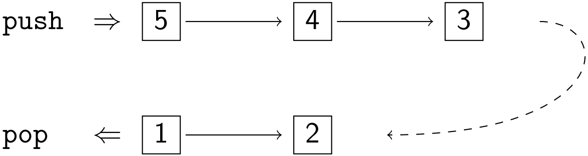
Breadth-First Traversal

usual iterator implemented using a **queue**

```
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
```

Persistent Queues

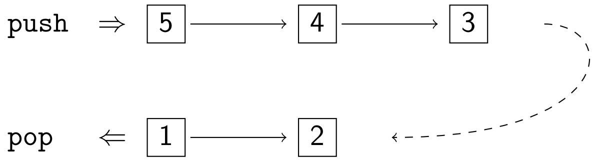
```
module Q = struct
  type α t = α list × α list
```




```
  let push = ...
  let pop = ...
  ...
end
```

Persistent Queues

```
module Q = struct
  type α t = α list × α list
```




```
  let push = ...
  let pop = ...
  ...
end
```

Persistent Iterator using a Persistent Queue

```
type enum = t Q.t

let start t = Q.push t Q.empty

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
```

Persistent Iterator using a Persistent Queue

```
type enum = t Q.t

let start t = Q.push t Q.empty

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
```

Persistent Iterator using a Persistent Queue

```
type enum = t Q.t

let start t = Q.push t Q.empty

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
```

What about the Zipper?

we need a Zipper generalized to **forests**

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

navigation

```
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"
```

What about the Zipper?

we need a Zipper generalized to **forests**

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

navigation

```
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"
```

Persistent Iterator using the Zipper

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

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

Persistent Iterator using the Zipper

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

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

Persistent Iterator using the Zipper

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

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

Persistent Iterator using the Zipper

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

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

Persistent Iterator using the Zipper

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

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

Persistent Iterator using the Zipper

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

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

Persistent Iterator using the Zipper

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

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

Persistent Iterator using the Zipper

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

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

Persistent Iterator using the Zipper

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

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

Simplification

the Zipper is actually always of the shape `Node(_,Top,_)`

we simplify:

```
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)
```

this is **exactly** the solution using persistent queues (which are here *inlined*)

Simplification

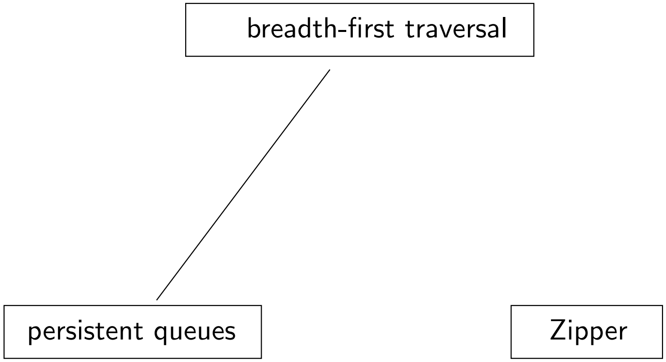
the Zipper is actually always of the shape `Node(_,Top,_)`

we simplify:

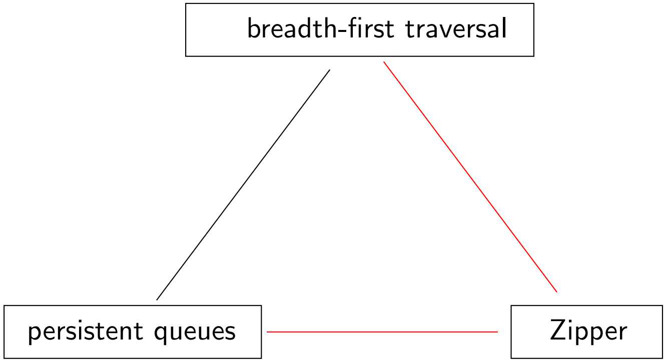
```
type enum = t list × t list
let start t = [], [t]
let rec step = function
  | [], [] → raise Exit
  | ll, [] → step ([], List.rev ll)
  | ll, E :: rr → step (ll, rr)
  | ll, N (l, x, r) :: rr → x, (r :: l :: ll, rr)
```

this is **exactly** the solution using persistent queues (which are here *inlined*)

Connections



Connections



Performances

traversal	code	random	left	right	full
inorder	ad-hoc	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	ad-hoc	1.01	0.76	0.10	0.26
	zipper	0.99	0.99	0.11	0.27
	lazy list	1.51	0.14	0.16	0.41
postorder	ad-hoc	1.26	0.86	1.04	0.28
	zipper	1.42	1.08	1.06	0.35
	lazy list	1.63	1.44	1.21	0.43
bfs	manual	14.57	0.44	0.47	4.62
	zipper	15.38	0.18	0.17	3.65

Performances

traversal	code	random	left	right	full
inorder	ad-hoc	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	ad-hoc	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	ad-hoc	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	manual	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

Back on Graph Coloring

in Ocamlgraph, persistent iterators are provided for depth-first and breadth-first traversals ⇒ direct application to graph coloring

```
let rec iterate iter =
  let v = Bfs.get iter in
  let m = G.Mark.get v in
  if m > 0 then
    iterate (Bfs.step iter)
  else begin
    for i = 1 to k do
      try try_color v i; iterate (Bfs.step iter)
      with NoColoring → ()
    done;
    uncolor v; raise NoColoring
  end
in
try iterate (Bfs.start g) with Exit → ()
```

Back on Graph Coloring

in Ocamlgraph, persistent iterators are provided for depth-first and breadth-first traversals ⇒ direct application to graph coloring

```
let rec iterate iter =
  let v = Bfs.get iter in
  let m = G.Mark.get v in
  if m > 0 then
    iterate (Bfs.step iter)
  else begin
    for i = 1 to k do
      try try_color v i; iterate (Bfs.step iter)
      with NoColoring → ()
    done;
    uncolor v; raise NoColoring
  end
in
try iterate (Bfs.start g) with Exit → ()
```

Back on Graph Coloring

Application

solving the **Sudoku** as a 9-coloring of a graph

tests show that

- a breadth-first traversal is really more efficient than a depth-first traversal
- persistent iterator are efficient (sudoku solved in 0.2 s on the average)

Back on Graph Coloring

Application
solving the **Sudoku** as a 9-coloring of a graph

tests show that

- a breadth-first traversal is really more efficient than a depth-first traversal
- persistent iterator are efficient (sudoku solved in 0.2 s on the average)

Back on Graph Coloring

Application

solving the **Sudoku** as a 9-coloring of a graph

tests show that

- a breadth-first traversal is really more efficient than a depth-first traversal
- persistent iterator are efficient (sudoku solved in 0.2 s on the average)

Conclusion

- persistent iterators should be part of standard libraries
- the Zipper is a useful tool to build such iterators
- as a bonus, we rediscovered the two-lists implementation of persistent queues

- what about other data structures?

Conclusion

- persistent iterators should be part of standard libraries
- the Zipper is a useful tool to build such iterators
- as a bonus, we rediscovered the two-lists implementation of persistent queues

- what about other data structures?