The ins and outs of iteration in *Mezzo*

François Pottier
francois.pottier@inria.fr

Jonathan Protzenko
jonathan.protzenko@inria.fr

Armaël Guéneau
armael.gueneau@ens-lyon.org

INRIA & ENS Lyon

HOPE'13

What is *Mezzo*?

Mezzo is a new programming language in the spirit of ML.

Mezzo's type system allows reasoning about state and state change.

It does so by keeping track of ownership via a mechanism of affine permissions.

Expressing state

We are interested in expressing object protocols, which present an inherent notion of state, in *Mezzo*.

Our case study, iteration over a collection:

- involves relatively simple protocols;
- illustrates how *Mezzo* expresses transfers of ownership.

Outline

Algebraic data structures

Higher-order iteration

Tree iterators as an abstract data type

Generic iterators as objects

Algebraic data structures

A mutable tree

```
data mutable tree a =
  Leaf
| Node { left: tree a; elem: a; right: tree a }
```

After this declaration:

- The algebraic type "`tree a`" is defined
- It will appear in permissions of shape "`t @ tree a`", for some term `t`

Permission analysis is flow-sensitive: different permissions will be available at different points of the program.

A mutable tree

The permission "`t @ tree a`" represents:

- Structural information: `t` is a tree with elements of type `a`
- Ownership information: we possess `t` and its elements

It can be seen as a token that grants access to `t` with type `tree a`.

Without this permission, you *cannot* access `t`.

A mutable tree

One can also write so-called structural permissions:

```
t @ Leaf
t @ Node { left: tree a; elem: a; right: tree a }
```

Permission refinement

```
match t with
| Leaf ->
    ...
| Node { left; elem; right } ->
    ...
end
```

Permission refinement

t @ tree a

```
match t with
| Leaf ->
    ...
| Node { left; elem; right } ->
    ...
end
```

Permission refinement

```
match t with
| Leaf ->
    ...
| Node { left; elem; right } ->
    ...
end
```

t @ Leaf

Permission refinement

```
match t with
| Leaf ->
  ...
| Node { left; elem; right } ->
  .
end
```

t @ Node { left: tree a; elem: a; right: tree a }

Permission refinement

```
match t with
| Leaf ->
  ...
| Node { left; elem; right } ->
  .
end
```

```
t @ Node { left: (=l); elem: (=x); right: (=r) }
* l @ tree a
* x @ a
* r @ tree a
```

Remark: “*” is separating.

Permission refinement

```
match t with
| Leaf ->
    ...
| Node { left; elem; right } ->

end
```

```
t @ Node { left = l; elem = x; right = r }
* l @ tree a
* x @ a
* r @ tree a
```

Remark: “*” is separating.

Recursive functions on trees

```
val size: [a] tree a -> int
```

- **`size`** requires an argument **`t`**, along with the permission "**`t @ tree a`**".
- **`size`** returns a value **`n`**, and produces the permission "**`n @ int * t @ tree a`**"

The input permissions of a function are returned, unless the keyword "**`consumes`**" is used.

A `size` function

```
val rec size [a] (t: tree a) : int =
  match t with
  | Leaf ->
      0
  | Node { left = l; right = r } ->
      size l + 1 + size r
  end
```

A `size` function

```
val rec size [a] (t: tree a) : int =

  match t with
  | Leaf ->
      0
  | Node { left = l; right = r } ->
      size l + 1 + size r
  end
```

t @ tree a

A `size` function

```
val rec size [a] (t: tree a) : int =

  match t with
  | Leaf ->
      0
  | Node { left = l; right = r } ->
      size l + 1 + size r
  end
```

```
t @ Node { left = l; elem = x; right = r }
* l @ tree a
* x @ a
* r @ tree a
```

A `size` function

```
val rec size [a] (t: tree a) : int =

  match t with
  | Leaf ->
      0
  | Node { left = l; right = r } ->
      let n1 = size l in
      let n2 = size r in
      n1 + 1 + n2
  end
```

```
t @ Node { left = l; elem = x; right = r }
* l @ tree a
* x @ a
* r @ tree a
```

Type- and permission-checking is a forward, step-by-step analysis.

Higher-order iteration

A higher-order iteration function

```
val     iter : [a, s: perm] (
  f: (a | s) -> bool,
  t: tree a
| s) -> bool
```

A call `f x` requires the permission `(x @ a) * s` and returns it.

Similarly, a call `iter(f, t)` requires and returns `(t @ tree a) * s`.

`iter` is polymorphic in `s`, which represents the effect of `f`.

A higher-order iteration function

```
val rec iter   [a, s: perm] (
  f: (a | s) -> bool,
  t: tree a
| s)  : bool =

  match t with
  | Leaf ->
      true
  | Node ->
      iter (f, t.left) && f t.elem && iter (f, t.right)
  end
```

On the way to iterators

Our `iter` function is easy to write and easy to use.

However, approaches where control is inverted, like iterators, are sometimes necessary, e.g., to solve the "same-fringe problem".

Tree iterators
as an abstract data type

Tree iterators, ADT style

Let's start with an OCaml implementation.

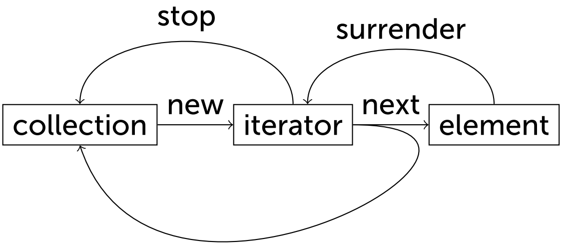
We wish to define:

- a data type **`tree_iterator`**;
- a **`new_iterator`** function: creates an iterator from a tree;
- a **`next`** function: produces a new element, if there is one.

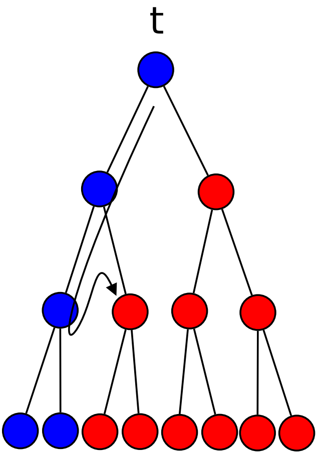
OCaml implementation

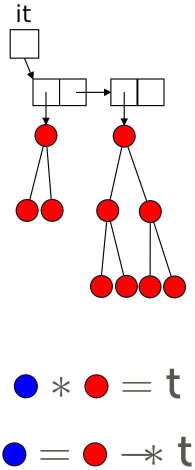
```
type 'a tree_iterator = 'a tree list ref

let new_iterator (t: 'a tree) =
  ref [t]

let rec next (it: 'a tree_iterator) : 'a option =
  match !it with
  | [] -> None
  | Leaf :: ts -> it := ts; next it
  | Node (l, x, r) :: ts -> it := l :: r :: ts; Some x
```

Iterator's object protocol



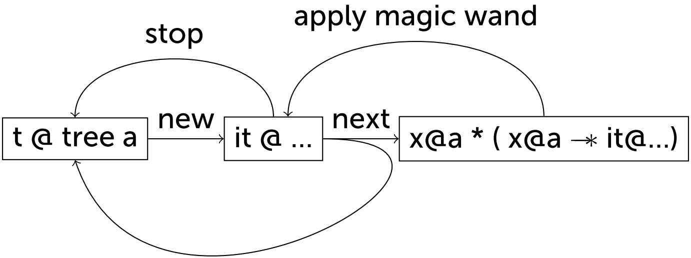
Permissions in the iterator





$\mathtt{P} \mathrel{-\!\!*} \mathtt{Q}$: a one-off permission to trade $\mathtt{P}$ for $\mathtt{Q}$.

Permissions through the protocol



Simulating the magic wand

In *Mezzo*, a function can be called as many times as one wishes (if suitable arguments and permissions are provided).

Yet, one can define a type of “one-shot functions”:

```
alias osf a b =
  {ammo: perm} (
    (consumes a | consumes ammo) -> b
  | ammo)
```

Simulating the magic wand

A **wand** is a one-shot function that deals only with permissions:

```
alias wand (pre: perm) (post: perm) =
  osf (| pre) (| post)
```

A function of type "**wand pre post**" is a one-shot opportunity to convert **pre** to **post**.

A typical use of the magic wand

```
alias focused a (post: perm) =
  (x: a, surrender: wand (x @ a) post)
```

This is a pair of a value **x** of type **a** and a unique opportunity to convert "**x @ a**" to **post**.

An interface for tree iterators (1/2)

The type of iterators is parameterized by a permission **`post`**, which is consumed by **`new`** and recovered via **`stop`**.

```
abstract tree_iterator a (post: perm)

val new: [a]
  (consumes t: tree a) ->
  tree_iterator a (t @ tree a)

val stop: [a, post: perm]
  (consumes it: tree_iterator a post) -> (| post)
```

`stop` does nothing at runtime.

An interface for tree iterators (2/2)

`next` queries the iterator for a new element.

```
val next: [a, post: perm]
  (consumes it: tree_iterator a post) ->
  either (focused a (it @ tree_iterator a post))
         (| post)
```

It returns either:

- an element **`x`** of type **`a`**, and the ability to recover "**`it @ tree_iterator a post`**" by abandoning "**`x @ a`**".
- **`post`** because the iterator has stopped (no more elements).

Implementation

The concrete type of tree iterators is almost as simple as in OCaml:

```
alias tree_iterator a (post: perm) =
  ref (focused (list (tree a)) post)
```

Unfortunately, the code (omitted) is a lot more verbose:

- magic wands must be explicitly constructed and invoked;
- existential packages must often be explicitly constructed.

Generic iterators as objects

Generic iterators: motivation

We want to be able to write code that uses "an iterator", instead of "a tree iterator" or "a list iterator"...

We define an object-oriented iterator: an object with **`next`** and **`stop`** methods.

Generic iterators

```
data iterator a (post: perm) =
  {s: perm}
    Iterator {
      next: (| consumes s) -> either (focused a s)
                                     (| post);
      stop: (| consumes s) -> (| post)
    | s }
```

The abstract permission s represents the internal state of the iterator.

ADT $\rightarrow$ OO conversion

We can “wrap” our ADT-style tree iterator as a generic OO-style iterator.

In that case, the witness for **`s`** is “**`it @ tree_iterator a post`**”.

Generic functions on iterators

Many standard stream operations can be defined on **`iterator`**.

For example, **`filter`** transforms an iterator into a new iterator.

```
val filter [a, s: perm, post: perm] (
  consumes it: iterator a post,
  f: (a | s) -> bool
| consumes s) -> iterator a (s * post)
```

Conclusion

Summary

Things we are happy with:

- *Mezzo* can express ownership transfers
- `iter` is easy to write, easy to use
- *Mezzo* can express simple object protocols

Things we are not so happy with:

- Too many type annotations are needed in the code
- Our iterator protocol is somewhat inflexible
- Will this scale to more complex protocols?

Related work

Design Patterns in Separation Logic, N. R. Krishnaswami et al.

Implements iterators in separation logic with a more precise analysis:

- Multiple iterators on one collection
- Updating the collection invalidates any existing iterator

They use a rich higher-order separation logic.

Prospects

- Add a builtin notion of “ghost” function, which would be erased at runtime
- Improve the type inference
- See how other objects protocols can be expressed in *Mezzo*