The ins and outs of iteration in *Mezzo*

Armaël Guéneau
ENS Lyon & INRIA
armael.gueneau@inria.fr

François Pottier
INRIA
francois.pottier@inria.fr

Jonathan Protzenko
INRIA
jonathan.protzenko@ens-lyon.org

Abstract

This is a talk proposal for HOPE 2013. Using iteration over a collection as a case study, we wish to illustrate the strengths and weaknesses of the prototype programming language *Mezzo*.

1. Introduction

Mezzo [2] is a high-level programming language in the style of ML. It is equipped with a strong static discipline of duplicable and affine permissions, which controls aliasing and ownership, and rules out certain mistakes, such as representation exposure and data races. In this talk, we would like to illustrate how *Mezzo* expresses transfers of ownership: sometimes easily, sometimes less so. We use iteration, a surprisingly rich problem, as a case study.

2. Algebraic data structures

Thanks to algebraic data types, it is easy to define list- and tree-like data structures. For instance, here is a type of mutable binary trees:

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

As in ML, a tree carries a tag, which identifies it as a leaf or a node. Unlike in ML, a type is interpreted not just as a structural description, but also as an assertion of ownership. Thus, when one writes "`t @ tree a`", this does not mean that "`t` is a tree (now and forever)". Instead, this means "`t` is a tree (now) and I have exclusive permission to read and write it". We say that "`t @ tree a`" is an affine permission: it is a unique token that grants access to `t` as a tree. A function that takes an argument `t` and wishes to access it as a tree requests this token from its caller and usually returns it to its caller. Permissions exist at type-checking time and incur no runtime overhead.

The permission "`t @ tree a`" can be refined, by analysis of `t`, into the permission "`t @ Node { left: tree a; elem: a; right: tree a }`", which itself is automatically split by the type-checker into a conjunction of four permissions:

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

(where `l`, `x`, `r` are fresh auxiliary names). Conjunction is naturally separating, so the left and right subtrees must be disjoint: this really is a type of trees, not of arbitrary graphs.

When reasoning abstractly, the permission "`x @ a`" is considered affine. This allows the type variable `a` to be later instantiated with an affine type, i.e., one that has a non-trivial interpretation in terms of ownership. For instance, "`t @ tree (ref int)`" means that `t` is a tree of pairwise distinct integer references, and represents the ownership of the tree and of its elements.

3. Higher-order iteration

The type-checker can split permissions (as above), join them, and set them aside when they are not needed (a "frame rule"). This makes it easy to write a recursive function that descends into a tree. For instance, the type of the "tree size" function is:

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

(Square brackets denote universal quantification.) By convention, this means that the call "`size t`" requires the permission "`t @ tree a`" and returns it. It is equally easy to write a higher-order function that descends into a tree and invokes a client-supplied function at every node:

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

The function `f` has access to one tree element at a time: it receives a permission of the form "`x @ a`" and must return it. Thus, this element is temporarily "borrowed" from the tree. The function `f` cannot access the tree, since it does not receive a permission for it. The universal quantification over a permission `s`, which `f` receives and returns, and which `iter` also receives and returns, allows the client to supply a function `f` that has a side effect on an area of memory represented by `s`. The Boolean value returned by `f` indicates whether iteration should continue (i.e., `false` represents an early termination request). The Boolean value returned by `iter` indicates whether iteration went all the way to the end (i.e., `false` means iteration was terminated early).

4. Tree iterators as an abstract data type

The higher-order function `iter` offers a style of iteration where the provider invokes the consumer when an element is available. In contrast, some programming languages, such as Java, encourage a style where the consumer invokes the provider in order to obtain an element. In *Mezzo*, at present, it is possible to encode this idiom, but this requires a deep understanding of the system. Expressing the iterator *interface* is tricky, because:

1. the permission for the collection (here, a tree) must disappear while the iterator is active, and must somehow be recovered once the iterator is discarded;
2. the permission for an element that was yielded by the iterator must be surrendered before the iterator can be queried again.

Furthermore, writing an iterator *implementation* is tricky, because:

3. whereas the function `iter` relies on an implicit control stack, an iterator contains an explicit representation of this stack, whose shape must be described by an appropriate permission—typically, some kind of "tree segment".

An iterator interface, in the form of an abstract data type (ADT) equipped with a number of operations, can be expressed as follows.

The type `tree_iterator` is parameterized with the type `a` of the elements and with a permission `post`, which represents the underlying collection. The idea is that this permission is recovered when the iterator is discarded.

```
abstract tree_iterator a (post: perm)
```

A tree iterator is created by invoking the function `new`:

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

When the type-checker examines the function call "`let it = new t in ...`", it checks that the permission "`t @ tree a`" is available at the program point before the call. At the program point after the call, this permission is gone (as specified by the `consumes` keyword), and the permission "`it @ tree_iterator a (t @ tree a)`" appears instead.

At any moment, one can discard the iterator and recover the permission `post` (which, in this context, is "`t @ tree a`") by invoking "`stop it`":

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

One may hope (and this holds in our implementation) that `stop` has no runtime effect and runs in constant time. It may be possible to extend *Mezzo* with a notion of "ghost" code and to declare `stop` as a ghost function.

While an iterator is active, it can be queried for a new element:

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

The call "`next it`" requires "`it @ tree_iterator a post`", and (perhaps surprisingly) consumes this permission, which means that, immediately after this call, the iterator can no longer be used. One must first examine the value returned by the call. Roughly speaking, either:

1. it carries an element `x` of type `a`, together with a promise that by abandoning the permission "`x @ a`", one can recover "`it @ tree_iterator a post`" and continue using the iterator; or
2. it carries the permission `post`, because the iterator has stopped.

(We omit the definition of the algebraic data type `either`, which represents a binary sum.) A value of type "`focused a post`" can be thought of as a dependent pair of a value `x` of type `a` and a "magic wand", that is, a "one-shot" ability to convert "`x @ a`" to `post`:

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

Mezzo does not currently have a primitive concept of a magic wand, viewed as a permission. As an approximation, we view a magic wand of `pre` to `post` as a runtime function that consumes `pre` and produces `post`. (Again, ideally, this should be a ghost function.) We make it a "one-shot" function (i.e., one that can be invoked at most once) by specifying that it consumes an abstract permission `ammo` and by pairing it with just one copy of `ammo`. (Curly braces denote existential quantification. An abstract permission is by default considered affine.)

```
alias wand (pre: perm) (post: perm) =
  {ammo: perm} (
    (| consumes (pre * ammo)) -> (| post)
  | ammo)
```

This concludes the definition of the iterator interface. We do not show the implementation of tree iterators, but note that the internal definition of `tree_iterator` itself relies on `focused`:

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

This means that a tree iterator is a stack of sub-trees and that, by abandoning the ownership of this stack, one recovers `post`, which represents the ownership of the complete tree.

5. Generic iterators as objects

We have constructed an abstract data type of iterators for a specific type of trees. The same approach can be applied to other data structures. Unfortunately, every time one does so, one obtains a new abstract data type of iterators. Thus, one cannot write generic code that uses "an iterator" without knowing how this iterator was constructed.

One way out of this problem is to adopt an object-oriented (OO) style and to define an iterator to be an object equipped with `next` and `stop` methods. The methods must have access to the iterator's internal state, which we represent by an abstract permission `s`. Thus, an iterator is a package of two functions that require `s` and of `s` itself:

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

alias iterator a (post: perm) =
  {s: perm} iterator_s s a post
```

This is an encoding in the style of Pierce and Turner, with the added twists that `s` is a permission, not a type (so the client never obtains a pointer to the object's internal state) and `s` is affine (so the client cannot invoke `stop` twice, for instance).

An ADT-style iterator can be converted a posteriori to OO-style. This is done by the following function, which accepts an iterator `it` of an arbitrary type `i`, provided this type is equipped with appropriate `next` and `stop` operations.

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

By combining `wrap` and the ADT-style library of the previous section (§4), one obtains:

```
val new_tree_iterator: [a]
  (consumes t: tree a) -> iterator a (t @ tree a)
```

Conversely, one can convert from OO style to ADT style, in the following sense: if desired, the type `iterator` defined above can be equipped with three operations `new` (a constructor), `next`, and `stop`, and made abstract.

An OO-style iterator is a stream (with mutable internal state), so it should not be surprising that many of the standard operations on streams can be defined on the type `iterator`. For instance, `filter` creates a new iterator out of an existing one:

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

A few points are worth noting:

1. The pre-existing iterator is consumed. It becomes owned by the new iterator, so to speak. Stopping the new iterator transparently stops the underlying iterator and yields `post`, which represents the ownership of the underlying collection(s).
2. The function `f` may have internal state, represented by the permission `p`, which `f` requires and returns. This permission is consumed by the call to `filter` (i.e., the new iterator takes possession of `p`), and is recovered when the new iterator stops.
3. `f` receives a permission to examine an element of type `a`, and must return this permission (there is no `consumes` keyword).

Other examples of operations that can be expressed include `map`, `zip`, `concat`, `equal` (which in the case of trees solves the "same fringe" problem), etc.

6. Turning a fold inside out

We have presented two approaches to iteration. In one, the producer is in control and invokes the consumer via a function call (§3); in the other, this situation is reversed (§4 and §5). The former approach makes it easy to implement a producer, while the latter approach facilitates life for the consumer, especially when one wishes to draw data out of several collections simultaneously.

In order to get the best of both approaches, one would like to be able to automatically derive a first-order iterator in the style of §4 and §5 out of a higher-order iteration function in the style of §3. It is well-known that this can in principle be achieved by using control operators. Unfortunately, for the moment at least, *Mezzo* does not have first-class control. Yet, as a preliminary study, we verify that an iterator can be derived out of a higher-order iteration function written in continuation-passing style (CPS).

In order to allow early termination, we use a "double-barreled continuation" style, and work with pairs of an abort continuation and a normal continuation. The type `continuations pre b1 b2`, defined below, represents such a pair. The abstract permission `ammo` is analogous to the one that was used in the definition of `wand` (§4) so as to ensure that a magic wand is applied at most once. Here, because both continuations require the same `ammo`, and because only copy of `ammo` is included, it ensures that at most one of the two continuations can be invoked.

```
alias continuations (pre : perm) b1 b2 =
  { ammo : perm } (
    failure: (| consumes (ammo * pre)) -> b1,
    success: (| consumes (ammo * pre)) -> b2
  | ammo
  )
```

The two continuations have the same domain: they expect zero runtime argument and they consume a permission `pre`, which is a parameter of this definition. They may have distinct answer types, `b1` and `b2`.

A CPS version of `iter` (§3) has the following type:

```
val cps_iter: [a, s : perm, b1, b2] (
  consumes t: (tree a | s),
  f: (
    consumes x: (a | s),
    consumes continuations (x @ (a | s)) b1 b2
  ) -> b2,
  consumes continuations (t @ (tree a | s)) b1 b2
) -> b2
```

The original `iter` (§3) does not consume "`t @ (tree a | s)`": it requires this permission and returns it to its caller. Here, this idea remains valid in principle. However, `cps_iter` does not return the permission "`t @ (tree a | s)`" to its caller: instead, it transmits this permission to one of its continuations. By the same token, the callback function `f` requires the permission "`x @ (a | s)`" and transmits this permission to one of its continuations.

Using `cps_iter`, one can re-implement `new_tree_iterator` (§5). The idea is to apply `cps_iter` to a function `yield` that captures the current continuation pair and stores it within the iterator. We omit the (fairly interesting) details. It is worth noting that this construction is independent of trees, so, by abstracting over "`tree a`" and `cps_iter`, it can be turned into a re-usable library.

7. Other approaches to iteration

One particularly elegant approach, well-known in the functional programming community, is to view the producer as a function that returns a lazy stream of elements, while the consumer is a function that accepts such a stream. In this approach, both producer and consumer are written in direct style, and the transfer of control is implicit. In *Mezzo*, by building upon a primitive notion of lock, one can define a type "`thunk a`" of suspensions, and on top of that, a type "`stream a`" of lazy streams. Suspensions and streams are considered duplicable, which means that they can be shared without restriction, just as in ML or Haskell. This approach works well, but is restricted to the case where the type `a` is itself duplicable.

Another attractive approach consists in running the producer and consumer as two threads connected by channels. The communication protocol is as follows. The producer sends an element `x` of type `a` along one channel, and the consumer replies (once it is done processing this element) by sending a message of type `(| x @ a)` (i.e., no runtime value, and the permission for `x`) along a second channel. Thus, the reply channel must be heterogeneous: every message has a different type, as it concerns a different element `x`. A limited kind of heterogeneous channel can be axiomatized in *Mezzo*, but this topic deserves further study: taking inspiration from Villard *et al.*'s work [3], we would like to understand how to best express complex communication protocols in *Mezzo* and how to extend *Mezzo*'s formal proof of type soundness with these features.

8. Conclusion

In this talk proposal, we have refrained from including numerous citations. However, there is a rich literature on type systems for mutable state and on approaches to iteration. Krishnaswami *et al.*'s paper [1] is particularly relevant. We will draw a comparison with some of the related work during the talk.

The "little" problem of iteration is surprisingly rich, because it involves the two fundamental issues of modularity and transfer of ownership. We believe that it is a good way of illustrating *Mezzo*'s expressive power and, more generally, how one might program in a language equipped with a pervasive notion of permission.

References

[1] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse. Design patterns in separation logic. In *Types in Language Design and Implementation (TLDI)*, pages 105–116, 2009.

[2] François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *International Conference on Functional Programming (ICFP)*, 2013. To appear.

[3] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 5904 of *Lecture Notes in Computer Science*, pages 194–209. Springer, 2009.