Mezzo: an experience report

François Pottier

Inria Paris

Lyon, December 2017

At the present time I think we are on the verge of discovering at last what programming languages should really be like. [...] My dream is that by 1984 we will see a consensus developing for a really good programming language [...]

Donald E. Knuth, 1974.

What is Mezzo?

A *programming language proposal*, in the tradition of ML.

Mainly Jonathan Protzenko's PhD work (2010-2014).

Try it out in your browser:

```
http://gallium.inria.fr/~protzenk/mezzo-web/
```

Or install it:

```
opam install mezzo
```

Joint work with:
Jonathan Protzenko, Thibaut Balabonski,
Henri Chataing, Armaël Guéneau, Cyprien Mangin.

- Agenda
- Design principles
- Illustration (containers; locks)
- Thoughts

Premise

The types of OCaml, Haskell, Java, C#, etc.:

- describe the *structure* of data,
- but say nothing about *aliasing* or *ownership*,
  - they do not distinguish *trees* and *graphs*;
  - they do not control who has *permission* to read or write.

Goals

Could a more ambitious static discipline:

- *rule out* more programming errors
- and *enable* new programming idioms,
- while remaining reasonably *simple* and *flexible*?

Goal 1 – rule out more programming errors

Classes of errors that we wish to rule out:

- *representation exposure*
  - leaking a pointer to a private, mutable data structure
- *concurrent modification*
  - modifying a data structure while an iterator is active
- *violations of object protocols*
  - writing a write-once reference twice
  - writing a file descriptor after closing it
- *data races*
  - accessing a shared data structure without synchronization

Goal 2 – enable new programming idioms

Examples of idioms that we wish to allow:

- *delayed initialization*
  - “null for a while, then non-null forever”
  - “mutable for a while, then immutable forever”
- *explicit memory re-use*
  - using a field for different purposes at different times

Design constraint – remain simple and flexible

Examples of design constraints:

- types should have *lightweight syntax*
- *limited, predictable* type annotations should be required
  - in every function header
- types should not influence the *meaning* of programs
- type-checking should be *easier than program verification*
  - use dynamic checks where static checking is too difficult

Non-goals

Mezzo is intended to be a *high-level* programming language.

Examples of non-goals:

- to squeeze the last bit of *efficiency* out of the machine
- to control *data layout* (unboxing, sub-word data, etc.)
- to *get rid* of garbage collection
- to express *racy* concurrent algorithms

- Agenda
- Design principles
- Illustration (containers; locks)
- Thoughts

Key design decisions

We have a limited “complexity budget”. Where do we spend it?

In Mezzo, it is spent mostly on a few key decisions:

- *replacing* a traditional type system, instead of *refining* it
- adopting a *flow-sensitive* discipline
- keeping track of *must-alias* information

Details of these key decisions:

- there is no such thing as *"the"* type of a variable
- at each program point, there are *zero, one, or several permissions* to use this variable
  - `b @ bag int`
  - `l @ lock (b @ bag int)`
  - `l @ locked`
- *strong updates* are permitted
  - `r @ ref ()` can become `r @ ref int` after a write
- permissions can be *transferred* from caller to callee or back
- permissions are *implicit* (declared at function entry and exit)
- if `x == y` is known, then `x` and `y` are interchangeable

Down this road, ...

After these bold initial steps, *simplicity* is favored everywhere.

Design decision – just two kinds of permissions

A type or permission is either *duplicable* or *unique*.

- immutable data is duplicable: `xs @ list int`
- mutable data is uniquely-owned: `r @ ref int`
- a lock is duplicable: `l @ lock (r @ ref int)`

No fractional permissions.

No temporary read-only permissions for mutable data.

The system *infers* which permissions are duplicable.

Design decision – implicit ownership

A type describes *layout and ownership* at the same time.

- if I (the current thread) have `b @ bag int`
  then I know `b` is a bag of integers
  and I know I have exclusive access to it

No need to annotate types with owners.

No need for "owner polymorphism" – type polymorphism suffices.

Design decision – lightweight syntax for types

A function receives and returns *values and permissions*.
A function type `a -> b` can be understood as sugar for

```
(x: =x | x @ a) -> (y: =y | y @ b)
```

By convention, *received* permissions are considered *returned* as well, unless marked consumed. The above can also be written:

```
(x: =x | consumes x @ a) -> (y: =y | x @ a * y @ b)
```

Design decision – lightweight syntax for types

A function that *"changes the type"* of its argument can be described as follows:

```
(x: =x | consumes x @ a) -> (| x @ b)
```

or, slightly re-sugared:

```
(consumes x: a) -> (| x @ b)
```

A result of type `()` is returned, with the permission `x @ b`.

Design decision – no loops

We encourage writing *tail-recursive functions* instead of loops.

Melding two mutable lists:

```
val rec append1 [a]
  (xs: MCons { head: a; tail: mlist a },
   consumes ys: mlist a) : () =
  match xs.tail with
  | MNil  -> xs.tail <- ys
  | MCons -> append1 (xs.tail, ys)
  end
```

Look ma, *no list segment*.

The list segment “behind us” is “framed out”.

Design decision – a static/dynamic tradeoff

Adoption & abandon lets one permission rule a *group* of objects.

- adding an object to the group is statically type-checked
- taking an object out of the group requires proof of membership in the group,
- which is verified *at runtime*,
- therefore *can fail*

This keeps the type system simple and flexible.

It is however *fragile*, and mis-uses could be difficult to debug.

- Agenda
- Design principles
- **Illustration (containers; locks)**
- Thoughts

A typical container API

Here is a typical API for a "container" data structure:

```
abstract bag a
val new:     [a] () -> bag a
val insert:  [a] (bag a, consumes a) -> ()
val extract: [a] bag a -> option a
```

Notes:

- The type `bag a` is unique.
- The type `a` can be *duplicable or unique*.
- `insert` *transfers the ownership* of the element to the bag; `extract` transfers it back to the caller.

Here is a typical API for a "container" data structure:

```
abstract bag a
val new:     [a] () -> bag a
val insert:  [a] (bag a, consumes a) -> ()
val extract: [a] bag a -> option a
```

Notes:

- `let b = new() in ...` produces a permission `b @ bag a`, *separate* from any prior permissions; thus, a "new" bag.
- `insert` and `extract` request and return `b @ bag a`, which tells that they (may) have an *effect* on the bag.
- *No null pointer, no exceptions.* We use options instead.

A pitfall

Because mutable data is uniquely-owned, *“borrowing”* (reading an element from a container, without removing it) is restricted to *duplicable* elements:

```
val find:
  [a]
  duplicable a =>
  (a -> bool) -> list a -> option a
```

This affects user-defined containers, arrays, regions, etc.

A possible workaround

Unrestricted borrowing *can* be expressed as a higher-order function:

```
val find: [a, b, p: perm]
  (a -> bool, list a, (option a | p) -> b | p) -> b
```

It is used as follows:

```
(* f @ ref int -> bool * xs @ list (ref int) *)
let y = find (f, xs, fun (x : option (ref int)) : int =
  (* ... use the element x  ... *)
  (* ... but cannot use xs! ... *)
) in
(* f @ ref int -> bool * xs @ list (ref int) * y @ int *)
```

Sugar and *stronger type inference* would be needed to simulate borrowing in the style of Rust.

The lock API is borrowed from concurrent separation logic.

A lock protects a fixed permission `p` – its *invariant*.
A lock can be *shared* between threads:

```
abstract lock (p: perm)
fact duplicable (lock p)
```

A unique token `l @ locked` serves as proof that the lock is held:

```
abstract locked
```

This serves to prevent double release errors.

The lock API

The invariant `p` is *fixed* when a lock is created.

It is *transferred* to the lock.

```
val new: [p: perm] (| consumes p) -> lock p
```

Acquiring the lock produces `p`. Releasing it consumes `p`. The data protected by the lock can be accessed *only in a critical section*.

```
val acquire: [p: perm]
  (l: lock p) -> (| p * l @ locked)

val release: [p: perm]
  (l: lock p | consumes (p * l @ locked)) -> ()
```

A typical use of the lock API

The lock API introduces *"hidden state"* into the language.

```
val hide : [a, b, s : perm] (
  f : (a | s) -> b (*          "f" has side effect "s" *)
| consumes s       (*    the call "hide f" claims "s" *)
) -> (a -> b)      (*             and yields a function *)
                   (* which advertises no side effect *)
```

Here is how this is implemented:

```
val hide   [a, b, s : perm] (
  f : (a | s) -> b
| consumes s
)  : (a -> b)
=
  (* Allocate a new lock. *)
  let l : lock s = new () in
  (* Wrap "f" in a critical section. *)
  fun (x : a) : b =
    acquire l; let y = f x in release l; y
```

- Agenda
- Design principles
- Illustration (containers; locks)
- Thoughts

In retrospect – did we get carried away?

The type system is “simple” and has *beautiful metatheory* (in Coq).

The early examples that we did *by hand* were *very helpful* but gave us a *false feeling* that type inference would be easy, which it is not:

- first-class universal and existential types, as in System F
- intersection types
- rich subtyping
- must perform frame inference, abduction, join

Type errors are *very difficult* to explain, debug, fix.

Safe *interoperability with OCaml* is a problem.

Type inference problems – example 1

The system can express *effect polymorphism*.

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

At a call site, must infer how to instantiate `p`.

Type inference problems – example 2

The system can express *one-shot functions*.

- `{p : perm} ((| consumes p) -> () | p)`
- no need for multiple ad hoc function types

Must infer where to "pack" and how to instantiate `p`.

Type inference problems – example 3

The system can express *intersection types*.

- `f @ t1 -> u1 * f @ t2 -> u2`
- this actually arises in our iterator library
- unexpected

At a call site, must infer which view of `f` to use.

Type inference problems – example 4

The system can *decompose / recompose* a view of memory.

- `x @ ref int` is interconvertible with
  `{y : term} (x @ ref (=y) * y @ int)`

Must infer where and how to recompose.

Conclusion

We got early *peer pressure* to formalize the metatheory.

- this helped us better *understand* and *simplify* Mezzo
- but *took manpower away* from implementation and evaluation

Designing a new type theory, as opposed to refining ML:

- seemed *more radical*, therefore appealing
- perhaps a *mistake?*
    - separating type- and permission-checking might be easier
    - and would permit interoperability with OCaml