A few lessons from the Mezzo project

François Pottier[1] and Jonathan Protzenko[2]

1 **INRIA**
`francois.pottier@inria.fr`
2 **Microsoft Research Redmond**
`jonathan.protzenko@ens-lyon.org`

Abstract

With *Mezzo*, we set out to design a new, better programming language. In this modest document, we recount our adventure: what worked, and what did not; the decisions that appear in hindsight to have been good, and the design mistakes that cost us; the things that we are happy with in the end, and the frustrating aspects we wish we had handled better.

Keywords and phrases static type systems; side effects; aliasing; ownership

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 A word about *Mezzo*

Mezzo is a programming language in the tradition of ML, whose type system incorporates a notion of ownership. The type system is close to separation logic: at each program point, a set of *permissions* describe what fragment of the heap "we" (the current procedure on the current thread) have access to and in what ways we can affect this fragment without violating what "others" (the callers of the current procedure on the current thread, as well as concurrently executing threads) assume about it. The type system of *Mezzo* has been mechanically proved sound [4, 1]: well-typed *Mezzo* programs are *memory-safe* and *data-race free*. A comprehensive paper about *Mezzo* is in submission [2]; bold and daring readers may also wish to read the second author's thesis [14].

Mezzo enables new programming patterns: thanks to its powerful ownership discipline, *type-changing updates* are permitted. This yields great flexibility: for instance, typestate, which is usually expressed using extra predicates that refine types [6], is now seen as a particular case of type-changing updates. Furthermore, a combination of singleton types and structural types allows the type system to track local aliasing relations. To keep the complexity of the type system under control, we deliberately do not support certain advanced ownership disciplines (such as static regions with multi-focusing; fractional permissions; etc.). In order to partially make up for the lack of these advanced features, we encourage the programmer to rely on dynamic tests when dealing with complex aliasing of mutable data. To that effect, *Mezzo* provides a novel adoption/abandon mechanism that allows a dynamic test to yield a static permission. *Mezzo* also provides locks, which are another dynamic mechanism for (temporarily) obtaining a static permission. In practice, adoption/abandon and locks are typically used in concert.

Mezzo does not just live on paper: one very explicit goal was to design a language *for programmers*, not just a stack of Greek symbols in a conference paper. To that effect, we split the language in several layers (one for the programmer, a lower-level one for the type-checker, and a kernel one for the proof of soundness). Furthermore, we designed type inference algorithms and wrote an implementation, which can be tried out online [15]. Finally, the *Mezzo* repository [13] contains several thousand lines of examples, made up mostly of data structures and algorithms.

2 Some fundamental design decisions

The *Mezzo* project started with the PhD of the second author, whose title was "towards better static control of side-effects". Our first decision was to design a new language, rather than build upon an existing one. A related decision was to focus exclusively on the surface language and on its type system. After type erasure, *Mezzo* is essentially a subset of OCaml, so a (well-typed) *Mezzo* program can be compiled and executed by translation to untyped OCaml (i.e., OCaml with unsafe type casts).

Starting afresh offers many benefits. The feature set of a fresh language is minimal. We did not have to adapt our type discipline to deal with objects, modules, or GADTs, to cite only a few of OCaml's features. The implementation of a fresh language is simple: we did not have to implement *Mezzo*'s type-checker as an extension of OCaml's type-checker, which would have been technically very difficult. The design of a fresh language can be radical: we did not have to retain compatibility with an existing code base. In fact, we were at liberty to deeply alter the way programmers think about types. Instead of viewing types as descriptive (and unalterable), we incorporate ideas from separation logic, we think of types as descriptive and prescriptive at the same time (and allow them to change as the program runs). Thus, we must abandon Hindley-Milner type inference, and replace it with a type-checking and type inference algorithm which in many ways resembles forward symbolic execution [5] and abstract interpretation.

Focusing on a functional core influenced the design of the language. In *Mezzo*, algebraic data types are fundamental. They can express state change, temporarily broken invariants, refined predicates, aliasing relationships... Had we chosen to construct an object-oriented language, the core building blocks of the language would have been objects instead.

Starting anew also has several drawbacks. Perhaps the biggest one is the absence of a conversion path from OCaml to *Mezzo*. Even though untyped *Mezzo* is (a subset of) untyped OCaml, there is no gradual way of porting a program from OCaml to *Mezzo*: the entire program must be altered so as to abide by *Mezzo*'s type discipline. Furthermore, we do not have a good way of allowing interoperability between OCaml and *Mezzo*. Although the function type and the `ref` type in OCaml and in *Mezzo* look superficially similar, they have different meaning: it is unsafe to naively let a mutable object (or a function that accesses a mutable object) cross the boundary. Perhaps one could restore safety by inserting dynamic checks at the boundary; this seems nontrivial and was left as future work.

For a research project that lacks the resources to go mainstream, the lack of interoperability is perhaps tolerable. We do feel, however, that providing a conversion path would have helped us write more examples and receive more user feedback. New programming languages that aim at seizing a market share do deal with interoperability: TypeScript has "definitions" for adding a typed interface on top of existing JavaScript code; Rust can be made ABI-compatible with C libraries, as long as suitable Rust types are provided for C functions.

3 Growing the language

One of our starting points was theoretical work by Charguéraud and by the first author [7]. We originally focused on the type system of a core language, leaving the design of a surface language to a later stage. This approach turned out to be flawed. Lacking any impetus from sample programs, we had a hard time figuring out where to push the design of the core language. Lacking any connection with the surface, we had an even harder time figuring out how inference could possibly work. We ended up entangled in technicalities and unable to form a grander vision of where we should go.

Things improved when we decided to drop this approach and instead work on the surface language first. One of the best steps we took at this point was to write several sample programs, even before the language existed. Designing a new language is a broad, ill-defined task. Writing fictitious programs, finding out exactly how we would like them to look and feel, and understanding what guarantees we would like the type system to provide, proved highly beneficial. This activity allowed us to understand early on that some of the features that we had in mind (such as static regions) were not practical; it also allowed us to imagine several novel features, such as adoption/abandon, a more flexible replacement for static regions.

3.1 Syntax, syntax, syntax

We spent a surprising amount of time working out the details of the syntax. This may seem futile; yet, a concise and natural syntax is a requisite for wide adoption, and a sign that the underlying concepts are simple.

Consider the type of the list `length` function. This type must express that the caller loses ownership of the list while the `length` function operates over it and, once the function returns, the caller regains ownership. Here is what this type looks like in the internal language of the type-checker:

$$\forall(a : \mathsf{type}), \forall(l : \mathsf{value}), (=l \mid l \,@\, \mathsf{list}\ a) \to (\mathsf{int} \mid l \,@\, \mathsf{list}\ a)$$

Thanks to our syntactic sugar, the user simply writes:

```
val length: [a] list a -> int
```

This simplification is obtained in two steps. First, in the surface syntax, we adopt the convention that a permission that appears in the domain of a function type also appears (implicitly) in its codomain. This means that $l \,@\, \mathsf{list}\ a$ can be omitted in the right-hand side of the arrow. Second, because the right-hand side no longer refers to the name l, it is no longer necessary to introduce this name. The universal quantification over l disappears, and the left-hand side of the arrow becomes just $\mathsf{list}\ a$.

In contrast, the type of list concatenation must express the fact that this function consumes the ownership of its two arguments. In the surface syntax, this can be done in any one of the following ways:

```
val append: [a] (consumes (list a, list a)) -> list a
val append: [a] (consumes list a, consumes list a) -> list a
val append: [a] (consumes xs: list a, consumes ys: list a) -> zs: list a
```

To allow the user to express fine-grained control over which arguments are lost and which are preserved, the **`consumes`** annotation can be nested arbitrarily deep within structural (tuple and data) types. The carefully-chosen convention that “permissions are preserved unless otherwise specified” has proven, in our experience, to save a lot of annotations.

Our last example is the `iter` function on lists. Its type differs from its classic OCaml counterpart in that the client function `f` may impose a type-changing update on the list elements. This type illustrates the flexibility of the **`consumes`** keyword: here, the permission for `xs` is consumed, while the permission for `f` is preserved. This type also illustrates a situation where the function’s codomain `(| xs @ list b)` refers to `xs`, a name for the first component of the function's argument. This exploits the "name introduction" construct `x: a`, which again can be nested at arbitrary depth within a structural type.

```
val iter : [a, b] (
  consumes xs: list a,
  f: (consumes x: a) -> (| x @ b)
) -> (| xs @ list b)
```

The type of `f` says: "Give me exclusive control over my argument `x` at type `a`. I will return to you a unit value, along with exclusive control of `x`, now at type `b`". The type of `iter` has a similar reading: it requires unique ownership of `xs` at type `list a` and returns unique ownership of `xs` at type `list b`.

We did not come up with this all at once, naturally. Our functions initially took multiple arguments. While trying to work out the translation of our conventions for function types into more atomic constructs, we realized that we could replace multiple-argument functions with single-argument functions and simplify the whole translation process. The functions `append` and `iter` above expect just one argument, which is a pair. This view is made possible by the fact that the `consumes` annotation and the name introduction construct `xs: ...` can appear inside a structural type. This design simplified the translation from surface syntax to internal syntax, made the surface syntax more regular, and augmented the expressive power of our function and structural types.

Incidentally, the tuple type `(x: t, y: u)` is not a primitive dependent tuple. It is an ordinary tuple whose components contain name introduction constructs. It is desugared using existential quantification and singleton types. The desugaring is symmetric, so both `t` and `u` may refer to both `x` and `y`. There is no left-to-right bias.

3.2 Algebraic data types are central

One thing that we believe is a strength of our type system is the unified vision of structure and ownership that it provides. For instance, we do not need to distinguish a type environment and a state environment that would store information such as "this list cell is uninitialized" or "this cell is initialized and will no longer be mutated". Instead, everything is expressed in the type. A typical way to encode state information about a memory block is to exploit algebraic data types, as follows.

```
data mutable cell =
  Cell { head: (); tail:     () }
data list a =
  Cons { head:  a; tail: list a } | Nil

val () =
  let x = Cell { head = (); tail = () } in
  (* x is uninitialized and has type: Cell { head:  (); tail:        () } *)
  x.head <- 0;
  x.tail <- Nil;
  (* x has been mutated and has type: Cell { head: int; tail:       Nil } *)
  tag of x <- Cons;
  (* x has been frozen and has type:  Cons { head: int; tail:       Nil } *)
  (* which may be folded back to:     Cons { head: int; tail: list int } *)
  (* which may be folded back to:     list int                           *)
```

Structural types provide a natural, integrated mechanism for tracking the state of a memory block, via its *tag*, or data constructor. By referring to the data type definitions, the type-checker knows that a memory block whose type is `Cell { ... }` is mutable (and uniquely owned), while a memory block whose type is `Cons { ... }` is immutable (and shared). In a structural type, the types of the fields are arbitrary: they need not match the types that appear in the data type definition. However, a structural type can be folded back to a nominal type, as in the last line above, only if the types of the fields do match the definition.

The above example shows a variety of features that we believe are novel. (1) We conflate products and sums in data type definitions. This comes at no runtime cost (in the OCaml heap, every block carries a tag anyway) and in our experience improves readability. (In contrast, OCaml does not allow naming the arguments of a data constructor.) (2) The tag update instruction adds a little bit of low-level expressive power (at the surface level, OCaml does not allow mutating the tag of a memory block). (3) Furthermore, as shown above, tag update can express freezing: by definition, the tag `Cell` implies mutability, while the tag `Cons` implies immutability. (4) Structural types allow keeping precise track of field updates: the type of a field can change at every update. This is sound because mutable memory blocks are uniquely owned. (5) A `match` construct refines a nominal type to a structural type, henceforth allowing field access. For instance, if `xs` has type `list t`, then a *Mezzo* programmer may choose to write:

```
match xs with Cons -> f xs.tail | Nil -> ... end
```

In contrast, an OCaml programmer would have to bind the tail of `xs` to a fresh variable as part of the pattern: `match xs with Cons (_, tail) -> f tail | Nil -> ... end`. In *Mezzo*, both styles are permitted.

Separation logic [16] and alias types [17] have shown that it is important to keep track of must-alias relationships in order to get an accurate description of the heap. Consider a variable `x` which has type `Cons { head: a; tail: list a }`. If one reads the `head` field by writing `let y = x.head in ...`, who "owns" the first list element at type `a`? Is it still owned by `x`, so to speak, or is it now owned by `y`? Rather than introduce a concept of "data type with a hole in it", or a concept of "borrowing", we choose to rely on *singleton types*. A singleton type `=x` has only one inhabitant, namely `x` itself. In the above situation, the *Mezzo* type-checker automatically introduces two auxiliary names `h` and `t` and makes the following statements:

```
  x @ Cons { head: =h; tail: =t }
  h @ a
  t @ list a
```

Then, upon finding the definition of `y`, the type-checker adds the statement that `y` has type `=h`, which one may write either `y @ =h` or in the form of an equation:

```
  y = h
```

The above four statements, or *permissions*, imply that `x.head` and `y` are aliases and can be used interchangeably. Furthermore, permissions are ownership assertions. The permission `h @ a` represents the ownership of a heap fragment whose root is `h` and whose extent is described by the type `a`. Similarly, `t @ list a` represents the ownership of a heap fragment rooted at `t`. Because a singleton type implies no ownership, the equation `y = h` claims no ownership. For the same reason, the permission `x @ Cons { head = h; tail = t }` represents the ownership of just one block of memory, at address `x`, and describes its contents in an exact way. In short, types and permissions have both layout and ownership readings.

All of this shows that structural types and singleton types are central concepts in *Mezzo*: together they convey not only shape information ("`x` is the address of a `Cons` cell"), but also must-alias information ("its head is `h`") and ownership information ("we own the memory block at address `x`"). Naturally, these ideas are inspired by Alias Types [17] and by Separation Logic [16]. We mesh them with algebraic data types, so that they integrate well with the language and allow expressing a variety of key programming patterns in a natural way.

For this approach to work, the system must have rules to decompose and recompose types, so as to switch back and forth between a coarser view where "`x` has type `list a`" and a lower-level, finer-grained description in terms of singleton types. Furthermore, the type-checker must be able to transparently apply these rules whenever necessary. During the implementation of the type-checker, we found this a challenge.

3.3 Ownership is central

Mezzo must support type-changing updates, not only because we are interested in typestate-checking as a high-level goal, but also because (as shown above) the manner in which we exploit singleton types essentially leads *every* update to be a type-changing update. However, a strong update is sound only if the modified object is uniquely owned. Therefore, an ownership discipline is required.

Turning this argument around, one may accept ownership as the primary concept, and view mutability as a consequence of it. *Because* we have exclusive access to a memory area, it is safe to change its contents in an arbitrary way. This cannot break what others assume about this area, because in fact they cannot assume anything about it; and this cannot cause a race condition, because they cannot access this area.

One way of understanding ownership is in terms of permissions. "We" are granted the right to perform certain operations ("we own this data structure exclusively, hence we may read and write it") whereas "others" are denied the right to perform certain operations ("we own this block exclusively, hence others cannot read or write it"). Another way of understanding ownership is in terms of knowledge. Exclusive ownership of a data structure means "we" know that this data structure exists, whereas "others" don't; shared ownership means "we" know about this data structure and "others" may know about it too. Naturally, the two views are dual, and either of them defines the other. If one thinks primarily in terms of permissions, then our "knowledge" is whatever assertion is stable in the face of permitted interference by others. Conversely, if one thinks primarily in terms of knowledge, then we have "permission" to perform whatever action preserves the knowledge of others. This dual understanding of ownership is reminiscent of rely-guarantee [10]. The classic type disciplines of C, Java, OCaml, etc., are a special case where every object is considered potentially shared and (as a consequence) every update must be type-preserving.

Anecdotal evidence suggests that ownership is a good concept for *users*, as it helps them have a mental model of what's legal and what's not, of what may happen and what may not happen, but also for *language designers*, as it helps figure out at an intuitive level whether a proposed typing rule makes sense in terms of ownership. One such mechanism is adoption and abandon.

3.4 Flexible ownership via adoption and abandon

Adoption/abandon works, in essence, as follows. If one owns an object `x` at type `t`, one may relinquish ownership of `x` and **`give`** it to an adopter `y`. One can think of `y` as maintaining a runtime list of its adoptees, which it owns. (Our implementation uses a more efficient scheme, where an adoptee points to its adopter). Adoption consumes the unique permission `x @ t`: the type system no longer keeps individual track of `x`. (It also consumes `x @ unadopted`: this guarantees that every object has at most one adopter, a condition that our implementation scheme requires.) Instead, the aggregate permission `y @ adopts t` represents the ownership of all adoptees of `y` as a group. While `x` is adopted, it no longer has type `t`. Still, it has type `adoptable`, which means it is a valid address. The address `x` can be copied without restriction: this allows mutable data to become aliased, and still remain usable, as described now. When one wishes to regain ownership of `x`, one may attempt to **`take`** `x` from `y`. This operation checks, at runtime, that `x` currently appears in the list of adoptees of `y`, and takes it out of this list. This yields the permissions `x @ t` and `x @ unadopted` again. The point of the runtime check is to prevent the duplication of these permissions.

`x` has type	**We know that...**	**Others know that...**	**mode**
`adoptable`	`x` is a valid heap address	`x` is a valid heap address	duplicable
`unadopted`	`x` is a valid heap address `x` is not currently adopted	`x` is a valid heap address	affine
`adopts t`	every adoptee of `x` has type `t`	–	affine

The ownership hierarchy was, initially, a purely static concept that was expressed in the types. Adoption and abandon allow part of this hierarchy to exist (and to be queried and modified) at runtime. The **`give`** and **`take`** operations allow move between the static and dynamic regimes of ownership.

This mechanism is perhaps one of the main contributions of *Mezzo*: it provides an *escape hatch* out of the purely static discipline. This allowed us to keep the type system relatively simple. If we had tried to statically keep track of complex ownership patterns, we would have exceeded our complexity budget.

Throughout the course of the *Mezzo* project, we had to navigate between two conflicting goals. On the one hand, we wanted a language that was more than a research project. For instance, a traditional affine (or linear) type system is not expressive enough for real-world programming: we had to allow, one way or another, a certain degree of aliasing. On the other hand, we wanted the system to remain usable by (skilled) programmers: we had to reject proposed features of the type system that we deemed too technical. Abandon and adoption was judged a good compromise.

3.5 Beyond duplicable versus affine?

The various *modes* of ownership were the topic of much discussion. For simplicity, *Mezzo* as it stands distinguishes only two modes, or two kinds of permissions, namely duplicable (i.e., shared) permissions and affine (i.e., non-duplicable, or exclusive) permissions. A duplicable permission can be viewed as affine; the converse is not permitted. Every permission is in fact affine.

One may think of classifying types using kinds, where a kind is "duplicable" or "affine". However, in order to type-check list `length`, one would then need kind polymorphism:

$$\mathsf{val\ length}\colon \forall\kappa, \forall(\alpha : \kappa), \forall(l : \mathsf{value}), (=l \mid l \,@\, \mathsf{list}\ a) \to \ldots$$

This additional layer of quantification seems too much of a burden. We prefer to view modes as predicates over types, much like type classes. Thus, the user can explicitly request that a type satisfy a certain mode:

```
val f: [a] duplicable a => (x: a) -> ...
val g: [a] affine a => (x: a) -> ...
```

Because every type is affine, the constraint `affine a` can in fact be omitted. This gives rise to concise types in many situations.

In theory, one could add other modes, such as "linear". Viewing an affine permission as linear would be permitted; the converse would be forbidden. In principle, this could help programmers ensure that resources (e.g., file descriptors) are properly freed. In practice, though, this would make the system more verbose: the constraint `affine a` would no longer be a tautology and might have to appear in many places. Furthermore, in order to prove a "complete collection" theorem (i.e., a linear object is never lost), one would have to impose a restriction: a resource guarded by a lock must be affine. (This is necessary because locks are never de-allocated.) This makes the idea less attractive: for instance, a lock cannot guard a file descriptor. Furthermore, even if "complete collection" holds, in practice, it seems that there are still (admittedly contrived) ways of getting rid of a linear object without properly freeing it; e.g., by converting it to the existential type `{a} (a | linear a)` and "giving" it to a "black hole" adopter that one threads through the entire program. For all these reasons, we left the exploration of "linear" (and other modes) to future work.

4 Looking at some code

Some programs can be expressed in a palatable manner using *Mezzo*. This is less true of some other programs. While there is no theorem classifying easy-to-write and hard-to-write programs, we have a few examples that are representative of the typical experience of writing *Mezzo* code.

4.1 Example #1: one-shot functions

A programming pattern that pops up quite often is that of "one-shot functions". Also known as linear arrows, these functions may only be called once. In *Mezzo*, by default, a function can only capture duplicable permissions, and is itself duplicable. Therefore, every *Mezzo* function may be called as many times as one wishes, provided of course that the caller is able to supply the permissions that this function requires. Nevertheless, one can simulate a one-shot function in *Mezzo*, as follows.

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

val promote [a, b, p: perm] (f: (consumes (a | p)) -> b | consumes p) :
  (| f @ osf a b) = ()
```

A one-shot function of `a` to `b` (where `a` and `b` are types) is a package of a function of `a | p` to `b` along with a single copy of `p`, where `p` is an opaque permission. That is, in order to invoke this function, one must supply not only an actual argument of type `a`, but also the permission `p`. Because `p` is existentially quantified (as indicated by the curly brackets), it is regarded as affine: that is, it cannot be duplicated. Because the first invocation of the function consumes `p`, any further invocation is forbidden. Thus, a function of type `osf a b` may be called at most once.

Creating a one-shot function should incur no run-time penalty. One can either use the `promote` function above, whose net effect is to perform an existential packing, and assume that the compiler will get rid of what is actually a no-op; or, one can write a `pack` instruction that bypasses the function call and performs the existential packing directly.

This encoding of one-shot functions may seem heavy, and the reader may wonder whether one could instead view affine functions as a primitive notion. We note that our approach is more general, as it allows encoding other related concepts, such an affine choice between two functions, e.g., "you may invoke either the success continuation or the failure continuation, but only one of them, and at most once".

The version of the one-shot function above takes and returns a run-time argument. We can define a restricted version of one-shot functions that only operates over permissions. The type below implements the "magic wand" of separation logic, written $p \multimap q$.

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

In this restricted version, the function takes and returns at run-time the unit value, but statically transforms the permission `pre` passed along with the argument into `post`.

These programming patterns would qualify as "easy to express" in *Mezzo*. They do sometimes require annotations, which can be made cumbersome by the fact that we need a `let flex` construct to refer to anonymous, existentially-quantified variables. That being said, we've been happy that these can be defined in the language, and don't have to be built-in features.

4.2 Example #2: strong updates

Code that performs strong updates is, in general, well-suited to the static discipline of *Mezzo*. A representative example is our implementation of "futures" (also known as "promises", or suspended computations).

A suspended computation works as follows: we allocate a mutable reference in the heap; we evaluate the computation and write its result into the reference; we *freeze* the reference (make it immutable) to express the fact that no further mutations shall occur.

In its initial state, a suspension is thus a standard *Mezzo* reference, defined as follows.

```
data mutable ref a =
  Ref { contents: a }
```

By performing a tag-update from `Ref` to `R` we can freeze a suspension into its final state, that is, into a `result a`.

```
data result a =
  R { result: a }
```

(From a more general perspective, tag mutations in *Mezzo* allow checking more invariants, as our type system is able to statically track this "freezing" pattern, where a data structure starts mutable, then is made immutable.)

There is a catch, however: we initially allocate a *mutable* reference, which is *affine*, hence not shareable. We do want to share it, however, as multiple clients will want to wait for the computation to finish in the background. We thus define the type `future a`. It is duplicable; this is the type the client manipulates, and that we export as `abstract future a` in the module's interface.

```
alias future a = (s: unknown, lock (s @ result a))
```

The type definition above is relatively concise: thanks to our binding rules, the name `s` may appear anywhere in the tuple. Quite surprisingly, the lock protects a duplicable permission, which may seem counter-intuitive. One way to intuitively understand the type `future a` is to think of it as a promise that, by the time the client acquires the lock, the reference will be frozen with the result of the computation.

The permission `s @ result a` is, naturally, not available initially; actually, we use the lock as a semaphore, as the code sample below demonstrates.

```
(* Creating a future from a one-shot function [k]. *)
val new [a] duplicable a => (consumes k: osf () a) : future a =
  (* The suspension starts out as a reference,
   * whose contents does not matter. *)
  let s = newref () in
  (* The lock is created in the [locked] state; the permission
   * [s @ result a] is *not* available yet! *)
  let l : (l: lock (s @ result a) | l @ locked)  = new_locked () in
  (* The computation that is spawned in the background. *)
  let compute (| consumes (k @ osf () a * s @ ref () * l @ locked)) : () =
    s := k();
    (* Turn [s @ Ref { contents: () }] into [s @ R { result: a }], that is,
     * into [s @ result a]. *)
    tag of s <- R;
    (* We can now release the lock for the first time. *)
    release l
  in
  (* Concurrently compute and return the future: *)
  spawn compute; (s, l)
```

Reading the result of the computation is then a mere matter of acquiring the lock; the first read will occur after the transition of the “semaphore” from a reference to a `result a`.

This flavor of programming, where a uniquely-owned data structure is mutated, then frozen, tends to work well in *Mezzo*. We have some flagship examples on lists: we can define tail-recursive versions of `map` and `concat`, where a new list cell is temporarily mutable, and is frozen (i.e., becomes immutable) once it has been fully initialized.

We also have the example of lazy thunks, where a combination of subtyping witnesses and existential quantification allows implementing thunks *in Mezzo* while ensuring that the type `lazy a` is covariant[1].

We agree that the suspensions example is somewhat technical; it seems to us, though, that it is still a strength of *Mezzo* that we can explain these precise mutations and ownership transfers within the type system. Doing the same in ML would require unsafe casts or heavy run-time checks.

[1] Unlike suspensions, the details are truly technical; the curious reader can read the `stdlib/lazy.mz` file in the source repository; it is heavily commented.

4.3 Example #3: mutable trees

Another kind of code that we have found to work well in *Mezzo* is imperative code without aliasing. Our implementation of mutable balanced binary search trees is representative of that class of programs, which in general includes all list-like and tree-like mutable data structures.

Our module for mutable trees represents about a thousand lines of *Mezzo* code. It adapts OCaml's tree library so as to allow in-place mutation. It does not use adoption/abandon.

```
data tree k a =
  | Empty
  | mutable Node {
      left: tree k a; key: k; value: a; right: tree k a; height: int
    }
```

We never mutate the Empty constructor; therefore, only the Node constructor needs to be mutable. Allowing Empty to remain immutable allows one to allocate a single value and use it subsequently in all places where an Empty constructor is required, thus acting like a null pointer.

The tree module features several internal functions; an interesting one is the function that re-balances a tree.

```
val bal: [k, a] (
  consumes t: Node {
    left: tree k a; key: k; value: a; right: tree k a; height: unknown
  }
) -> tree k a
```

This function type is interesting in several ways. First, it demands that the argument t be not just a tree k a, but specifically a Node. This kind of pre-condition would be expressed in OCaml as a comment; *Mezzo* enforces it via the static type-checking discipline. This pattern is one that we've come to use frequently.

Second, because the function writes the height field but does not read it, it does not need any hypothesis about the type of this field. We therefore document in the function type that the function will not read the height field.

Most of the functions in this module require a comparison function. Rather than requiring the user to provide a comparison function for each call to, say, find, we defined a new dependent type that bolts a specific comparison function onto the tree type.

```
data mutable treeMap k (cmp : value) a =
  TreeMap { tree: tree k a; cmp | cmp @ (k, k) -> int }
```

This is not a radically new theoretical feature, but rather another programming pattern that we have found convenient in many situations. In OCaml, one would either risk mixing a tree with an unrelated comparison function, or one would have to use functors. *Mezzo* offers a more light-weight mechanism for that.

4.4 Example #4: borrowing

Borrowing is a term that covers multiple issues. Let us tackle a specific one: assigning a type to the find function for lists. This raises a difficulty in terms of ownership: because `find` returns a pointer to a list element, it duplicates this element. (That is, the element is now accessible both via the list and as the result of `find`.) Hence, `find` must be restricted to lists of duplicable elements. How can one work around this restriction?

A higher-order version of `find` can be easily written, where the caller passes a function that describes what to do once the element is found.

```
val find: [a] (
  xs: list a,
  pred: a -> bool,
  f: (x: a) -> ()
) -> ()
```

This is not satisfactory, however, because it requires the user to change the control-flow. What we would like is for the caller to get a pointer into the list, and make sure the list is invalidated as long as the caller holds a pointer to the element. Thus, the element would be temporarily "borrowed" from the list.

The good news is, this ownership pattern *can* be expressed in *Mezzo*. The bad news is, it requires crafting some "ninja" *Mezzo* code that is outside the reach of a casual user. The signature of such a `find` function is as follows, where we reuse the `wand` type we saw earlier:

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

val rec find: [a] (consumes xs: list a, pred: a -> bool)
-> either
     (| xs @ list a)
     (focused a (xs @ list a))
```

The *focused* type describes the pair of an element, along with a wand that *consumes* the element, in exchange for a certain permission `post`.

The `find` function consumes ownership of the list, and either returns it immediately, meaning that the element was not found, or returns a *focused* element that, once "surrendered", grants ownership of the original list again. This signature hence expresses faithfully the protocol for ownership transfer that we outlined earlier.

Quite regrettably, the implementation of `find` is fairly difficult to comprehend, and proper explanations are outside the scope of this paper (they can be found in our journal paper [3]). Let us just mention briefly some ingredients:

- the identity function serves as the implementation of the magic-wand – in other words, we abuse a function that does nothing at run-time to encode invariants in the type system, for lack of ghost code;
- the inference engine of *Mezzo* cannot figure out how to synthesize the "magic wand" – therefore, the user must perform a manual `pack` instruction and provide the existential witness;
- in order do so, the user must be able to name the type variables that are automatically introduced by the type-checker when it "auto-unpacks" an existentially quantified type or permission. Currently, this is done by using a `let flex` construct which allows naming a type variable after has been introduced. Using this construct requires a rather low-level understanding of the type-checker.

Similar, if only greater difficulties are the heart of our work on iterators [9].

This latter example certainly falls in the category of programs that are harder to express in *Mezzo* than they ought to be. In the current state of things, we do not know how to make these programs with complex ownership protocols easier to write in *Mezzo*. There is, of course, the option of restricting `find` to duplicable elements. This may sound like a very special use-case, but in *Mezzo*, one can always convert affine elements to duplicable ones, using adoption/abandon:

- define a global adopter object,
- initially `give` all elements to the adopter,
- manipulate elements of type `dynamic` (duplicable),
- wrap every access to an element with a pair of `take` and `give`.

Therefore, a container of non-duplicable elements can always be viewed as a pair of a container of duplicable elements (adoptees) and an adopter. This is not satisfactory, though: the use of adoption/abandon is never desirable as it has a runtime cost and introduces a possibility of failures.

Taking a step back, other programming patterns are certainly hard to express in *Mezzo*, and if we were to extend the type system with more ad-hoc rules, we would certainly consider extra candidates. For instance, taking an immutable pointer to a mutable block (C++'s `const` modifier) is currently not supported by *Mezzo*; the right way to support this would certainly be fractional permissions.

5 Difficulties revealed by the implementation

Implementing *Mezzo* proved remarkably beneficial. This task allowed us to confirm and strengthen our basic intuitions about the type discipline. At the same time, it revealed many difficult problems that we initially did not clearly anticipate.

One particularly thorny issue is the "merge problem". Consider a `match` expression where `x` has type Nil at the end of one branch, and has type `Cons { head = h; tail: Nil }` at the end of the other branch, where `h` has type `a`. What is the type of `x` after the match expression? `list a` seems a natural answer. A closer look, though, shows that `list (=h)` is also a valid answer, although most certainly not the one the user had in mind! The problem becomes even more difficult in the presence of exclusively-owned data, and admits no principal solution in the general case.

Another thorny issue is inference of polymorphic instantiations. Suppose `x` has type `Cons { head = h; tail: Nil }` and `h` has type `a`. If one calls `identity x`, then the type-checker tries to infer a polymorphic instantiation. Several alternatives arise: `list a`, of course, but also `Cons { head = h; tail: Nil }`, `Cons { head: a; tail: Nil }`, and other variations, including the singleton type `=x` and the "top" type `{a} a`. Here, the final decision is innocuous: the identity function, after all, returns the ownership to the caller, so regardless of the instantiation choice, all is well. One can easily imagine non-trivial cases where the decision has dramatic consequences on the rest of the type-checking process.

As we saw, *Mezzo* can express "one-shot functions". This is nice, and can be extended to describe more complex situations, such as a one-off choice between several functions, or a function whose type changes after every call. Unfortunately, after a one-shot function has been called, its type lingers. The function still has type `(a | consumes p) -> b`, for some opaque permission `p`, and the type-checker cannot see that this makes the function useless. More generally, a function may have multiple types. *Mezzo* can express intersection types:

one can easily state that `x` has type `t` *and* type `u`. Intersection types can sometimes, but not always, be simplified. In particular, at a call site, if the function happens to have several types, it is not easy for the type-checker to determine which one applies; in fact, several of them could apply, and the choice could have consequences down the road!

Our work on iterators [9] stumbles upon all of the above difficulties. To summarize our current position: the type system is in theory remarkably powerful and regular, and can (at least in some cases) express complex protocols of ownership transfer. In practice, however, using the system requires a great deal of expertise and a non-trivial amount of annotations.

After all, this is not surprising: the *Mezzo* type-checker needs to somehow perform type inference in System F (a subset of *Mezzo*!), decide entailment in separation logic, and solve the frame inference problem (at function call sites). These problems are hard or undecidable. The problem is possibly made harder by the presence of (contravariant) function types, which are absent in first-order separation logics.

5.1 The state of the implementation

Our current implementation relies on heuristics and limited backtracking. The type-checker does not always terminate. The situations where it diverges are rare, but it is not completely clear how to rule them out. Most of the time, our heuristics work well. Nevertheless, they are unsatisfactory in principle, and difficult to maintain. Furthermore, should inference fail, the user is puzzled, and type errors are difficult to explain.

Perhaps a few examples may illustrate the kind of difficulties that we encounter in the type-checker. Many difficulties are related to the comparison of function types. For instance, the type-checker cannot prove that the following type is a subtype of itself.

```
val f: (| p * q) -> ()
```

The reason is, the algorithm for deciding subtyping compares functions; it then compares domains. In the process, the algorithm is faced, on the one-hand, with rigid variables `p * q` and, on the other hand, with flexible (unification) variables `?p * ?q`. Figuring out that `?p` ought to instantiate onto `p` and `?q` onto `q` amounts to a associativity/commutativity search. The algorithm does not perform this kind of search. More generally, because of our translation of external syntax into an internal one, a function may have several, equivalent types (with or without singleton types, for instance); the type-checking algorithm then needs to deal with a variety of equivalent representations seamlessly.

Other situations involve finding existential witnesses: a rule of our system allows instantiating a unification variable `a` into the conjunction of `b` (a type) and `p` (a permission), written `b | p`.

```
alias t1 = [a] () -> a
alias t2 = [p: perm] () -> (() | p)
```

The type `t1` is a subtype of `t2`: it suffices to instantiate `a` into `b | p`, then to instantiate `b` onto the unit type `()`. The type-checking algorithm is unable to figure that out, though: exploring the application of this typing rule would render the search space too big.

The picture depicted above is bleaker than it ought to be. In practice, a syntactic comparison makes sure that a type is always a subtype of itself (and provides, based on experimental measurements, a significant performance boost). Furthermore, the algorithm for comparing function types has been extensively fine-tuned and works most of the time. By "most of the time", we mean that over the 7,000 lines of "real" *Mezzo* that we have written (this includes blank lines and comments, but excludes files that are just unit test-cases), only 97 type applications appear, meaning roughly one annotation every 70 lines of code. We believe this to be an acceptable penalty.

Furthermore, after an internal presentation in the C++ group at Microsoft, it turned out that programmers there were absolutely unfazed at the prospect of a compiler that may fail, unpredictably, for some unknown reason, and may require them to provide more annotations. It seems that this is the common lot of a C++ programmer who uses templates, and the audience was unanimously happy to pay the unpredictability for more expressive power.

It thus seems that the situation is not as bad: failures of the type-checker are, after all, rare, and the user can always annotate herself out of a tricky type-checking situation.

6 The proof of Mezzo

The machine-checked definition of *Mezzo* is organized as a kernel, on top of which sit three (almost) independent extensions. The kernel calculus can be described as a concurrent call-by-value λ-calculus, equipped with an affine, polymorphic, value-dependent type-and-permission system. The extensions are: (a) strong (i.e., affine, uniquely-owned) mutable references; (b) dynamically-allocated, shareable locks; (c) adoption and abandon. We prove type soundness ("well-typed programs do not crash") and data-race freedom ("well-typed programs are data-race free").

The definitions and proofs add up to about 14K (non-blank, non-comment) lines of code. Out of this, a library for working with de Bruijn indices and a library for reasoning about separation, both of which are reusable, occupy about 2Kloc each. The remaining 10Kloc are split between the kernel (roughly 4Kloc) and its three extensions (roughly 6Kloc).

The current state of the formalization, with which we are fairly pleased, is the result of an iterative process. There were three main iterations.

The initial iteration was Pottier's proof of type soundness for a (sequential) affine type system with mutable state and hidden state [11].

The second iteration was a first version of the definition and proof of *Mezzo*. Compared to the previous iteration, a few important technical simplifications are worth mentioning. First, whereas the previous type system had (singleton and group) regions and a type for region inhabitants (e.g., "x has type $[\rho]$" means that x is an inhabitant of the region ρ), *Mezzo* has value-dependent singleton types (thus, "x has type $=x$"). Second, whereas the previous system required equi-recursive types (which play a role in the meta-theory of the anti-frame rule), *Mezzo* gets away with iso-recursive (i.e., algebraic) data types, whose theory of equality is significantly simpler. Third, whereas the previous proof relied on two calculi, connected by a rather tricky proof that "erasure is a simulation", we were able to work with just one calculus, where it is clear that permissions do not exist at runtime. A snapshot of the formalization of *Mezzo* at this point in time is given by an unpublished paper [12].

In the third and last iteration, we placed greater emphasis on the modularity of the formalization. In the previous iteration, the treatment of mutable memory blocks and that of adoption and abandon were intermingled. The type constructor for memory blocks served also as a type for adopters and as a type for adoptees; this led to a monster typing rule (rule Block [12, Figure 5]). In this iteration, instead, we introduced two new type constructors, adopts T and unadopted, which deal with these aspects, independently of the structure of memory blocks. This made the system more expressive (e.g., give and take can now be viewed as ordinary polymorphic functions, whereas their types previously could not be expressed). This also made the proof more modular: mutable references and adoption and abandon are now almost independent of one another. (They cannot be entirely independent, as adoption and abandon requires embedding an adopter pointer within every memory block.) Finally, we added concurrency and locks, which were not covered by the previous two iterations.

The current proof can be found online [1] and is described by a paper in submission [2]. One unsatisfactory aspect is that, although we would like to think of *Mezzo* as a kernel calculus plus three "almost" independent extensions, in reality the current Coq formalization is monolithic. Achieving true modularity in the syntax of the calculus, in its semantics, in its typing rules, and in the proofs, is difficult and still a research topic [8].

7 What now?

There remain a lot of open questions about *Mezzo* itself.

- Can we design, even if just on paper, a type-checking and type inference algorithm that is sound and complete for a subset of *Mezzo*?
- Can we work out a good mechanism for interacting with existing OCaml code bases?
- Can we offer a better user experience, especially in terms of type error messages?
- Do we need a wider variety of ownership mechanisms, as shown e.g. by Cyclone [18]?

If we were to re-think *Mezzo* along different lines, there are several paths we could explore.

- One might wish to layer a permission analysis on top of an existing typed language, such as a subset of OCaml. The system would be less expressive, but the analysis would be simpler, and violations would be easier to explain. *Mezzo*'s unification of types and permissions was more radical and in a sense more elegant, but this simpler approach is perhaps more reasonable.
- We could design a system that is unsound but still pragmatically useful. Take Dart, for instance; the language features mutable, covariant containers and takes responsibility for it. Yet, the mere fact that it checks lexical scoping of variables is a massive improvement over JavaScript. We could imagine a variant of *Mezzo* that features, for instance, OCaml's weak references without additional checks. This would be unsound, but would help convert existing programs.

Looking back on *Mezzo*, we remain happy with the design of the type system. We believe it is expressive, concise, and that the adoption/abandon mechanism strikes a nice balance between expressiveness and complexity. We also believe that reasoning in terms of ownership is natural for beginners. We do remain unsatisfied, though, with the shortcomings of our implementation, and with the lack of interoperability with existing OCaml code.

8 Bibliography

References

1 Thibaut Balabonski and François Pottier. A Coq formalization of *Mezzo*, take 2, July 2014. `http://gallium.inria.fr/~fpottier/mezzo/mezzo-coq.tar.gz`.

2 Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of Mezzo, a permission-based programming language. Submitted for publication, July 2014.

3 Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of Mezzo, a permission-based programming language. Submitted for publication, July 2014.

4 Thibaut Balabonski, François Pottier, and Jonathan Protzenko. Type soundness and race freedom for Mezzo. In *Proceedings of the 12th International Symposium on Functional and Logic Programming (FLOPS 2014)*, volume 8475 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2014.

5 Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

6 Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 301–320, 2007.

7 Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *International Conference on Functional Programming (ICFP)*, pages 213–224, 2008.

8 Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *Principles of Programming Languages (POPL)*, pages 207–218, 2013.

9 Armaël Guéneau, François Pottier, and Jonathan Protzenko. The ins and outs of iteration in Mezzo. Higher-Order Programming and Effects (HOPE), 2013. `http://goo.gl/NrgKc4`.

10 Cliff B Jones. Specification and design of (parallel) programs. In *IFIP congress*, volume 83, pages 321–332, 1983.

11 François Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of Functional Programming*, 23(1):38–144, 2013.

12 François Pottier. Type soundness for Core Mezzo. Unpublished, January 2013.

13 François Pottier and Jonathan Protzenko. *Mezzo*. `http://protz.github.io/mezzo/`, July 2014.

14 Jonathan Protzenko. *Mezzo: a typed language for safe effectful concurrent programs*. PhD thesis, Université Paris Diderot-Paris 7, 2014.

15 Jonathan Protzenko. *Mezzo*-web: try *Mezzo* in your browser, 2014.

16 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

17 Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381. Springer, 2000.

18 Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in Cyclone. *Science of Computer Programming*, 62(2):122–144, 2006.