



# Programming with permissions: the Mezzo language

Jonathan Protzenko, François Pottier

► **To cite this version:**

Jonathan Protzenko, François Pottier. Programming with permissions: the Mezzo language. ACM SIGPLAN Workshop on ML, Sep 2012, Copenhagen, Denmark. <hal-01092204>

**HAL Id: hal-01092204**

**<https://hal.inria.fr/hal-01092204>**

Submitted on 8 Dec 2014

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Programming with permissions: the *Mezzo* language

Jonathan Protzenko

Inria

*jonathan.protzenko@ens-lyon.org*

François Pottier

Inria

*francois.pottier@inria.fr*

*Mezzo* is a functional programming language with effects, in the tradition of ML. *Mezzo* aims to provide a successor to OCaml with a finer control of aliasing and effects. We offer stronger static guarantees on the mutable store: the language can express non-aliasing and separation properties. This fine-grained control of ownership and effects allows *Mezzo* to type-check programs previously deemed unsafe by the OCaml type-checker.

Idioms such as delayed initialization or strong (type-changing) updates are possible: *Mezzo* understands the in-place `swap` function, which swaps the two components of a pair. Similarly, a tail-recursive, destination-passing style version of `map` can be type-checked.

*Mezzo* is a language for users: our system tracks effects more accurately than global effect systems, but we use dynamic checks (section 3.1) to keep the system simple. This introduction is hardly self-contained, and the curious reader may wish to refer to [1] for a full introduction to the language.

## 1 Permissions

The key concept in *Mezzo* is the notion of permission. Permissions describe how objects are laid out in memory: they describe the shape of the heap. Permissions also enable programmers to control the ownership of objects, which turns out to be paramount in a concurrent setting.

A permission is written  $x @ t$ , meaning “ $x$  may be viewed with type  $t$ ”. A permission grants access to an object:  $x @ \text{ref int}$  asserts that “we” have the right to access  $x$ , and that it is a valid integer reference to boot. Permissions have no existence at runtime.

A permission may also assert what “others” (other threads, other parts of the program) are allowed to do, depending on the *mode* of the permission. *Duplicable* permissions describe read-only blocks in the heap; such permissions can be freely copied and passed to others – effectively sharing access to the block. *Exclusive* permissions describe read-write blocks in the heap; they cannot be copied, that is, the block has a unique owner.

	<b>duplicable</b>	<b>exclusive</b>
<b>we</b>	read-only	read-write
<b>others</b>	read-only	—

Figure 1: Access control

At any point in the program, a set (conjunction) of permissions is available; the set may include several permissions that refer to the same object.

### 1.1 Permission conjunction

The conjunction of two permissions  $p$  and  $q$  is written  $p * q$ . Our conjunction extends the separating conjunction of separation logic:

- on *exclusive* portions of the heap,  $*$  behaves as the usual separating statement:  $x @ t * x' @ t'$  asserts that  $x$  and  $x'$  cannot point to the same memory location – this is a *must-not-alias* constraint;
- on non-exclusive portions of the heap, the various permissions just need to be consistent with each other:  $x @ \text{int} * x @ \text{int}$  is such a consistent conjunction.

### 1.2 Manipulating permissions

Here is a datatype definition for duplicable pairs:

```
data pair a b = Pair { left: a; right: b }
```

`let x = Pair {left = 7; right = 3}` creates such a pair, yielding a new duplicable permission  $x @ \text{pair int int}$ . *Mezzo* automatically *unfolds* this permission and introduces fresh names for the components, thus yielding the following conjunction:

```
x @ Pair { left: =1; right: =r } *  
1 @ int * r @ int
```

The `left` field of  $x$  has the singleton type `=1`: this means  $x.\text{left}$  and `1` refer to the same object – this is a *must-alias* constraint. From now on, we will omit the colon and write `Pair { left = 1; right = r }`.

Should the programmer write `let y = x.left`, a new permission will be added, namely  $y @ =1$ , which we write  $y = 1$ . Thus, a conjunction of permissions can include a set of equations.

### 1.3 Example: swapping a pair in place

Objects of type `pair` are duplicable, hence immutable. Uniquely-owned, mutable pairs are defined as follows:

```
exclusive data xpair a b =  
  XPair { left: a; right: b }
```

The code from figure 2 demonstrates how to swap the two components of such a pair (comments indicate the set of available permissions after the line has been executed).

Notice the change in the permission of  $x$  between lines 2 and 3: it accounts for the new aliasing relationship, since the `left` field now points to `r` instead of `1`. Line 4 works in a similar fashion.

If we wish, we can trade the final conjunction for a weaker, more concise permission, namely  $x @ \text{xpair b a}$ . This code snippet changes the type of  $x$ : we call this a *strong update*. We'll see in the following section how to turn this code snippet into a function.

```

1 let x = XPair {left = ...; right = ...} in (* x @ XPair {left = l; right = r} * l @ a * r @ b *)
2 let t = x.left in (* x @ XPair {left = l; right = r} * l @ a * r @ b * t = l *)
3 x.left <- x.right; (* x @ XPair {left = r; right = r} * l @ a * r @ b * t = l *)
4 x.right <- t (* x @ XPair {left = r; right = t} * l @ a * r @ b * t = l *)

```

Figure 2: A fragment of code that performs a swap

## 2 Effects in function types

### 2.1 Some base types

*Tuple types* are written  $(t_1, \dots, t_n)$ . If  $z$  is the tuple expression  $(x, y)$ , then we have  $z @ (=x, =y)$ . Furthermore, if  $x @ t * y @ u$  is also available, then we can recombine these three permissions to obtain  $z @ (t, u)$ .

*Package types* allow one to bundle permissions along with a type: we write  $(t|p)$  to store permission  $p$  along with a value of type  $t$ . The left-hand-side exists at runtime; the right-hand side does not.

### 2.2 Some accurate function types

Let us now consider the simple case of `swap1`, a function that takes a pair and returns a new pair with its components swapped. The function does not perform an in-place update: it returns a fresh pair.

```

val swap1: pair a b -> pair b a
let swap1 (Pair { left = l; right = r }) =
  Pair { right = l; left = r }

```

Functions must be annotated with their signature in *Mezzo*. This signature may read like ML, but it conveys requirements about the permissions *consumed* and *returned* by the function. This naïve version of the swap function fails to type-check; let us now see why.

In order to call `swap1 x`, the permission  $x @ \text{pair } a \ b$  must be available to the caller, to be consumed by `swap1`. Moreover, we understand this permission to be *returned to the caller*. Therefore, `swap1 x` will return to the caller both the original permission on  $x$  and a new value with type `pair b a`.

However, after calling `swap1`, the components of the pair will be pointed to by both the old pair and the fresh one. Therefore, for the body of `swap1` to type-check, its signature must demand that  $a$  and  $b$  be duplicable types.

```

val swap: duplicable (a, b) =>
  pair a b -> pair b a

```

The body of the function will now type-check. Interestingly enough, the following signature is also valid:

```

val swap2: (consumes x: pair a b) -> pair b a

```

Here, the argument  $x$  is *consumed* by the function; that is, the permission referring to it is not returned to the caller. This type is more general: we leave it up to the caller to (implicitly) save a copy of the permission on  $x$ , by duplicating the permission before calling `swap2`, if possible.

Finally, if we want to write an in-place version of `swap` that operates on exclusive pairs, we need to express the fact that the function *modifies* the type of its argument.

```

val xswap: (consumes x: xpair a b)
-> ( | x @ xpair b a)

```

Here, `xswap` operates in place; therefore, it returns a tuple with no value components, i.e. the unit type. The permission on the argument is *not* returned; instead, a different permission for  $x$  is returned to the caller, reflecting the fact that the type of  $x$  has now changed.

## 3 Beyond simple permissions

### 3.1 An adoption mechanism for sharing

Basic permissions only allow for ownership trees; for mutable data structures with sharing, *Mezzo* uses a mechanism called *adoption*. It allows one to alias mutable objects, but uses runtime checks to ensure only one person at a time may “view” the mutable object. These dynamic checks may fail: we believe this is the price to pay to keep the system reasonably simple and expressive.

### 3.2 Concurrency

Permissions embody access control: only one thread at a time may own an exclusive permission, i.e. access a mutable location in the heap. Moreover, the adoption feature is thread-safe: several threads may try to acquire an object through the *abandon* operation, but runtime-checks make sure only one thread obtains an exclusive permission in the end. Finally, we provide a library for locks, exposed as duplicable objects that protect a permission. Therefore, programs written in *Mezzo* are data-race free.

The interaction of *Mezzo* and concurrency is still being worked on; we do not have sophisticated mechanisms such as fractional permissions yet, but this is an area of the design of *Mezzo* that is likely to evolve.

## 4 The current status of *Mezzo*

We are presently working on a prototype type-checker; the prototype can already type-check simple programs. Challenges include providing useful error messages, and elaborating heuristics for expressions with no principal type. In parallel, we are working on a formal proof of correctness for *Mezzo*, using the Coq proof assistant. We proved subject reduction for a core subset of *Mezzo*; we are looking forward to extend this proof to the whole language.

## References

- [1] François Pottier and Jonathan Protzenko. Programming with permissions: an introduction to *Mezzo* (long version). <http://gallium.inria.fr/~fpottier/publis/mezzo-tutorial-long.pdf>, September 2012.