View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Illustrating the *Mezzo* Programming Language

Jonathan Protzenko

INRIA
Rocquencourt, France
jonathan.protzenko@ens-lyon.org

Abstract

When programmers want to prove strong program invariants, they are usually faced with a choice between using theorem provers and using traditional programming languages. The former requires them to provide program proofs, which, for many applications, is considered a heavy burden. The latter provides less guarantees and the programmer usually has to write run-time assertions to compensate for the lack of suitable invariants expressible in the type system.

We introduce *Mezzo*, a programming language in the tradition of ML, in which the usual concept of a type is replaced by a more precise notion of a permission. Programs written in *Mezzo* usually enjoy stronger guarantees than programs written in pure ML. However, because *Mezzo* is based on a type system, the reasoning requires no user input. In this paper, we illustrate the key concepts of *Mezzo*, highlighting the static guarantees our language provides.

1998 ACM Subject Classification D.3.2 Applicative (functional) languages

Keywords and phrases Type system, Language design, ML, Permissions

Digital Object Identifier 10.4230/OASIcs.FSFMA.2013.68

1 Introduction

Type systems help programmers reason about the types of the manipulated objects, which embed information about their memory structure. Programs which obey a strong static discipline, such as that of ML, therefore have the powerful property that that they cannot go wrong. In other words, a well-typed program will not lead to a segmentation fault.

In practice, programmers want to reason beyond the memory layout of objects. Real-world objects often follow a protocol, going through different *states*, that only permit certain operations. A file descriptor starts *uninitialized*, then it may move to *ready*, before being *closed*. The "open" operation may only be performed on an uninitialized file descriptor, while the "close" operation only works when the file descriptor is ready. Thus, the file descriptor changes *states*, while preserving its *type*. However, traditional type systems fail to help programmers statically check *state* invariants.

Mezzo [6] is a programming language that reads and feels like ML, but that is equipped with a more powerful type system, which attempts to answer the above concerns. Since *Mezzo* has a more rigid typing discipline than ML, some programs that previously type-checked in ML will be deemed too unsafe. Conversely, as *Mezzo* allows more precise reasoning, some programs that previously could not be type-checked in ML will be understood.

In *Mezzo*, the notion of state and that of a type are conflated. An object which moves from a state to another is an object whose type changes. For instance, the "open" operation will change the type of its argument from *uninitialized* to *ready*. This design choice requires careful reasoning about *ownership*. Indeed, it is crucial that no other part of the system sees the object with its previous type, as this would naturally lead to an inconsistency, and protocol violations. Therefore, the type system should track ownership and avoid undesired aliases, as having two distinct names for the same object makes it difficult to ensure consistency.


© Jonathan Protzenko;
licensed under Creative Commons License CC-BY
1st French Singaporean Workshop on Formal Methods and Applications 2013 (FSFMA'13).
Editors: Christine Choppy and Jun Sun; pp. 68–73

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Literature offers a wealth of related work, and *Mezzo* draws inspiration from several areas. The biggest source of inspiration is *Separation Logic* [8, 4], a program logic that describes the state of the heap. In separation logic, asserting that an object has a given type amounts to owning that object. We reuse that principle, but turn it into a type system through our notion of permission, while also extending the reasoning to non-mutable portions of the heap. The *Plaid Project* [2] annotates references to objects with permission. This asserts both what one is allowed to do with the object, as well as what others may do with it. Our permission mechanism supports similar reasoning, but unifies both the state and the mutation invariants of an object, using permissions. In *Mezzo*, pointers can be copied, while the original permission on the object remains. We keep track of aliasing through the use of singleton types, inspired by *Alias Types* [9]. Our notion of affinity, expressing that items may be used at most once, while others may be used freely, is reminiscent of *Linear Types* [1].

We begin with an introduction to permissions, a core concept in *Mezzo*. Next, we discuss a case study, emphasizing several possible mistakes ruled out by our typing discipline. Finally, we give an overview of other *Mezzo* features and conclude with pointers to reference material.

2 An introduction to permissions

The central concept in *Mezzo* is that of a *permission*. While in the λ-calculus we say that "x has type t", in *Mezzo* we say we have permission x @ t, which we read "x at type t". We can think of a permission as a token that grants access to variable x with type t.

Unlike traditional typing judgements, permissions are transient. The user may possess x @ t at some point of the program, and have instead x @ u later on, which accounts for the fact that the type of x changed from t to u, i.e. that the *state* of x changed.

A permission may be obtained by creating a new value. Writing "`let x = (1, "hello")`" yields x @ (int, string), granting its owner the right to use x as a tuple of an int and a string.

2.1 Permissions control effects

Permissions appear in the signature of functions. Let us consider the type of the `length` function, which, as the name implies, computes the length of a list. The square brackets stand for universal quantification.

```
val length: [a] (x: list a) -> int
```

The introduction of the name x, along with its type list a, is syntactic sugar: the function expects an argument named x, along with a permission x @ list a. Conceptually, the function demands a token of ownership from its caller, so as to iterate over the list and compute its length. Hence, whenever one wishes to *call* the function on argument x, a permission x @ list a will be removed from the caller's current set of permissions.

Unless otherwise specified, and as a syntactic convention, we understand the permission x @ list a to be returned to the caller. Therefore, after the function call, the caller will regain x @ list a, along with a permission r @ int, where r is the name of the return value.

Another, more sophisticated function type, is that of the `annotate` function. It takes a *mutable* binary search tree of strings and modifies each node to store a pair, consisting of its original value and the size of the corresponding subtree, which the function returns.

```
val annotate: (consumes t: mtree string)->(int|t@ mtree (string,int))
```

Thanks to the `consumes` keyword, this function now takes a permission t @ mtree string from the caller and returns a *different* permission for t, namely t @ mtree (string, int). Therefore,

the type of t changes through a call to `annotate`. This is a *type-changing update*, which the permissions mechanism accurately describes.

2.2 Permission denote ownership

In order for the above function to be sound, no one else must own a copy of t @ mtree string, since that copy would be invalidated after calling `annotate`. Therefore, permissions that denote mutable portions of the heap must be uniquely owned. We say that the permission t @ mtree string is *exclusive*. The type system enforces this policy, by preventing exclusive permissions from being duplicated.

Conversely, x @ (int, string) denotes read-only knowledge, as our tuples, integers and strings are immutable. This knowledge is permanent, as the type of x will never change. Hence, it is sound to share that information. We say that the permission is *duplicable*, and the type-checker will allow the user to obtain as many copies of the permission as desired.

A permission x @ t therefore states not only that "x has type t", but also that "we own x at type t". The user (and the type-checker) can, by looking at t, infer whether t is exclusive or duplicable, i.e. whether they have an exclusive, read-write access, or a shared, read-only access to the object. The details for this procedure, called *mode inference*, are available [7].

Some permissions are neither exclusive nor duplicable; they are said to be *affine*. Such a permission is x @ a, where a is an abstract type variable, which may occur in the body of a function polymorphic in a. We have to be conservative and make no assumptions on a.

2.3 Permissions track aliasing

At any given program point, a current permission is available, granting us ownership of a part of the heap. Combining *atomic* permissions of the form x @ t into a composite permission is achieved using the $*$ connective; we say that the conjunction of p and q is $p * q$. This conjunction is reminiscent of separation logic. Indeed, if t and t' are both exclusive, the conjunction x @ $t * y$ @ t' implies that, because one cannot hold two exclusive permissions for the same variable, x and y must be distinct. This is a *must-not-alias constraint* and we state that our $*$ conjunction is separating on exclusive portions of the heap.

Moreover, $*$ extends the conjunction of separation logic to non-exclusive portions of the heap. If t' is duplicable, then x @ $t * y$ @ t' yields no information: x and y may or may not be aliases, and the conjunction just has to be consistent. The same situation holds if both t and t' are duplicable. Normally, conjunctions are consistent: if Nil denotes the empty list cell, x @ list int $*$ x @ Nil is a conjunction that is always consistent. However, inconsistent conjunctions exist, such as x @ mtree int $*$ x @ mtree int. Our system has been proved sound, meaning that a program cannot reach a configuration where this conjunction holds. This point in the program is unreachable: it is statically ruled out as "dead code".

These must-not-alias constraints, expressed implicitly in a conjunction, are completed by *must-alias constraints*, which are expressed using *singleton types*. A singleton type is of the form $=y$, where y is a program variable. This type has exactly one inhabitant: y itself. Therefore, having x @ $=y$ means that x and y are actually equal; in particular, if they are pointers, they point to the same value. We write this using syntactic sugar as $x = y$.

A singleton type appears whenever one creates an alias. If x @ t holds, then writing `let y = x in ...` yields a new permission $x = y$, without duplicating the original permission on x, which greatly simplifies reasoning. Singleton types are pervasive in *Mezzo*; they are particularly useful when combined with *structural types*.

Listing 1 Definition of mutable, binary trees

```
data mutable mtree a =
  | Null
  | Node { left: mtree a; value: a; right: mtree a }
```

Listing 1 above shows how to define an algebraic data type in *Mezzo*. Defining such a type allows one to obtain permissions of the form x @ mtree a. However, the type of x may be refined using a match expression; one may trade this permission for a more precise one, of the form x @ Node {left : mtree a; value : a; right : mtree a}. To understand what it means to own a value with such a type, let us rewrite this compact permission, introducing names for the three fields of x, as: x @ Node {left : $=l$; value : $=v$; right : $=r$} $*$ l @ mtree a $*$ v @ a $*$ r @ mtree a.

The ownership semantics of the compact permission can now be understood as stating that we own a memory block at address x of size four, containing a tag Node and three fields. We also own two mutable trees located at addresses l and r, along with a value of type a named v. The points-to relationships are represented by the singleton types. Similarly, the meaning of a nominal permission, such as x @ mtree a, is the disjunction of the meaning of its unfoldings x @ Null and x @ Node {left : mtree a; value : a; right : mtree a}.

3 A *Mezzo* case study

We now discuss a motivating example that, by using all the mechanisms described above, avoids several pitfalls. This example, as shown in listing 2, consists of splitting a mutable binary search tree. The function `split` "steals" the ownership of its argument t from its caller, and returns two binary search trees: the first one containing all values $v \leq k$ and the second one containing all values $v > k$. The `split` function abstracts over the comparison function `cmp`. We omit the re-balancing of the tree and focus on a self-contained example.

There are several pitfalls that await the programmer when writing such code. The user may inadvertently copy a key: this would violate the invariant that the two sub-trees form a partition. The user may return, as the right sub-tree, a pointer into the left sub-tree: this would create undesired sharing, leading to subtle bugs when two concurrent threads will want to access the two sub-trees, assuming that they are distinct. The permissions mechanism, by ensuring that exclusive knowledge cannot be duplicated, enforces these invariants statically.

Listing 2 In-place splitting of a mutable binary search tree

```
1  val rec split_right [a] (
2    consumes parent: Node { left: mtree a; value: a; right = child},
3    consumes child: mtree a,
4    k: a,
5    cmp: (a, a) -> int
6  ): (mtree a | parent @ mtree a) =
7    match child with
8    | Null -> Null
9    | Node ->
10       if cmp (child.value, k) <= 0 then
11         split_right (child, child.right, k, cmp)
12       else begin
13         let left_leq, left_gt = split (child.left, k, cmp) in
14         parent.right <- left_leq;
15         child.left <- left_gt;
16         child
17       end end
18
```

```
19 and split [a] (consumes t: mtree a, k: a, cmp: (a, a) -> int)
20   : (mtree a, mtree a) =
21   match t with
22   | Null -> Null, Null
23   | Node ->
24       if cmp (t.value, k) <= 0 then begin
25         let right_gt = split_right (t, t.right, k, cmp) in
26         t, right_gt
27       end else begin
28         let left_leq, left_gt = split (t.left, k, cmp) in
29         t.left <- left_gt;
30         left_leq, t
31       end end
```

3.1 The `split` function

The `split` function is the entry point. At the start of the function body, the permission is:

$$t \mathbin{@} \mathsf{mtree}\ a * k \mathbin{@} a * \mathsf{cmp} \mathbin{@} (a, a) \to \mathsf{int}$$

The function begins by matching on its argument t. If t is Null, two empty trees are returned. In the converse case, the permission on t is refined to:

$$t \mathbin{@} \mathsf{Node}\,\{\mathsf{left} : = l; \mathsf{value} : = v; \mathsf{right} : = r\} * l \mathbin{@} \mathsf{mtree}\ a * v \mathbin{@} a * r \mathbin{@} \mathsf{mtree}\ a$$

In the case that $t.\mathsf{value} \leq k$ holds, a call to `split_right`, whose meaning we will explain in the next section, is made. In the case that $k < t.\mathsf{value}$ (line 29), values greater than k may be found in the left sub-tree. The recursive call yields a partition of left sub-tree, consuming $l \mathbin{@} \mathsf{mtree}\ a$, while producing $\mathsf{left\_leq} \mathbin{@} \mathsf{mtree}\ a * \mathsf{left\_gt} \mathbin{@} \mathsf{mtree}\ a$. We re-attach values greater than k into $t.\mathsf{left}$, thus changing the permission of t into $t \mathbin{@} \mathsf{Node}\,\{\mathsf{left} : = \mathsf{left\_gt}; \mathsf{value} : = v; \mathsf{right} : = r\}$ Values lesser or equal to k are returned, along with t, which now contains the set of all possible values greater than k.

Which mistakes could an absent-minded programmer do? A first one would be forgetting to perform the assignment "`t.left <- left_gt`", at line 30. The permission for t would still be $t \mathbin{@} \mathsf{Node}\,\{\mathsf{left} : = l; \mathsf{value} : = v; \mathsf{right} : = r\}$. However, the call to `split`, at line 29, consumed the permission for l: it is no longer available, thus preventing this type from being folded back to $\mathsf{mtree}\ a$, when exiting the function. *Mezzo* would reject this program.

Another beginner's mistake would be to return the value `(left_gt, t)`. Following the return type `(mtree a, mtree a)`, *Mezzo* would consume $\mathsf{left\_gt} \mathbin{@} \mathsf{mtree}\ a$ to prove that `left_gt` is a tree. Next, *Mezzo* would have to prove that `t` is a tree, using permission $t \mathbin{@} \mathsf{Node}\,\{\mathsf{left} : = \mathsf{left\_gt}; \mathsf{value} : = v; \mathsf{right} : = r\}$. Specifically, this implies proving that `t.left`, also known as `left_gt`, is also a tree. Unfortunately, that exclusive permission was already consumed. This situation is therefore rejected.

3.2 The `split_right` function

The `split_right` function is written in a different style, as it takes a non-null `parent` tree, along with its right `child`. After the function call, the parent is still a tree, holding all values lesser or equal to k, while the returned tree contains all values greater than k.

The call to `split_right` at line 11 is legal, as we know that `child` is a `Node`, which justifies using it as the first argument. *Mezzo* statically checks that the second argument is indeed

the right child of the first: this information is known statically, due to the use of singleton types. This contrasts with the usage in traditional imperative languages, where typical code would rely on a loop and two mutable variables, with the implicit invariant that one is the right child of the other. Here, the invariant is made explicit and *Mezzo* can rule out misuses.

The type-checker applies recursive reasoning. After the call to `split_right` at line 11, if ret denotes the return value of the recursive call, the remaining permission is:

$$\mathsf{parent} \mathbin{@} \mathsf{Node}\,\{\mathsf{left} : \mathsf{mtree}\ a; \mathsf{value} : a; \mathsf{right} : {=}\mathsf{child}\} * \mathsf{child} \mathbin{@} \mathsf{mtree}\ a * \mathsf{ret} \mathbin{@} \mathsf{mtree}\ a$$

The type-checker then performs one last folding, to obtain the desired return type for the function. Note that the function call is tail-recursive, while the reasoning is *not*. Indeed, the use of recursive functions with distinct pre- and post-conditions yields more expressiveness than the use of traditional loops. This allows for stronger invariants.

4 Conclusion

Due to its *permissions* formalism, the *Mezzo* language manages to state precise invariants for programs that rely on mutable state, thus preventing several programming mistakes. The key mechanisms enforcing this rely on ownership, linearity and singleton types.

Permissions, as presented here, cannot account for non tree-shaped aliasing patterns. However, *Mezzo* offers several mechanisms for evading this restriction, e.g. locks, Boyland's nesting [3] and our own adoption/abandon mechanism. A more thorough discussion can be found [7], which details the language with typing rules and a formal definition of permissions. A gallery of programs along with extended material are available on the website [6].

The soundness of *Mezzo* has been machine-checked [5]. In the future, we wish to extend the language and its soundness proof with concurrency, to guarantee data-race freeness.

References

1 Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3: A linear language with locations. *Fundamenta Informaticæ*, 77(4):397–449, 2007.

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

3 John Tang Boyland. Semantics of fractional permissions with nesting. *ACM Transactions on Programming Languages and Systems*, 32(6), 2010.

4 Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

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

6 François Pottier and Jonathan Protzenko. *Mezzo*. `http://gallium.inria.fr/~protzenk/mezzo-lang/`, January 2013.

7 François Pottier and Jonathan Protzenko. Programming with permissions in *Mezzo* (long version). Unpublished, March 2013.

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

9 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.