The Mezzo language

François Pottier
francois.pottier@inria.fr

Jonathan Protzenko
jonathan.protzenko@inria.fr

INRIA

ML Workshop 2012
1 Mezzo pitch

2 A primer on permissions

3 A dynamic discipline of ownership

4 The current state of Mezzo
Plan

1. *Mezzo* pitch
2. A primer on permissions
3. A dynamic discipline of ownership
4. The current state of *Mezzo*
Mezzo is a **strict and impure** functional programming language; **Mezzo** offers a fine-grained control of **side-effects, aliasing and ownership**.
Mezzo strikes a balance between ease-of-use and complexity by combining a static ownership discipline with runtime tests.
Plan

1. Mezzo pitch
2. A primer on permissions
3. A dynamic discipline of ownership
4. The current state of Mezzo
My first permission!

Variables don't have types; there are permissions.

```ml
let y = ("foo", 3) in
```

This snippet generates a permission

```ml
  y @ (string, int)
```

One can think of it as a token that grants access to 
y with type (string, int).
Permissions do not exist at runtime.
val length: [a] (y: list a) -> int

- The argument has an (optional) name y.
- `length` requires a permission `y @ list a` and
- returns the very same permission: this is the default.
- The function also produces a value of type `int`. 
xswap swaps the two components of a mutable pair.

```ocaml
val xswap: [a, b] (consumes y: xpair a b) -> (|| y @ xpair b a)
```

- we introduce `y` as the name of the argument;
- the argument is consumed, i.e. `y @ xpair a b` is not returned;
- however, a new permission `y @ xpair b a` is returned instead.
Permissions can change!

Permissions replace types. At one point, we may have:

\[ y @ xpair a b \]

and later on, obtain:

\[ y @ xpair b a \]

Therefore, the set of available permissions may change with time.
A sound example?

This is how a permission can be traded for another one.

```plaintext
let y = e₁ in
(* y @ xpair a b *)
xswap y;
(* y @ xpair b a *)
e₂
```

For the `xswap` example to be sound, “no one else” should “see” `y`. This implies `xswap` should have exclusive access to its argument (no aliases).
A primer on permissions

Different *modes* for types

<table>
<thead>
<tr>
<th></th>
<th>duplicable</th>
<th>exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>me</em></td>
<td>read-only</td>
<td>read-write</td>
</tr>
<tr>
<td><em>others</em></td>
<td>read-only</td>
<td>---</td>
</tr>
</tbody>
</table>

- *xpair* is *exclusive*: it is mutable (read-write), and uniquely-owned, while
- *int*, *string*, are *duplicable*: they are immutable (read-only), and shared.
Permissions enforce access control

This means:

- `y @ int` can be duplicated, while
- `y @ xpair int int` cannot.
If $\tau$ is an exclusive type,

- $y @ \tau$ guarantees we own a memory block with type $\tau$;
- $y @ \tau * z @ \tau$ is a conjunction that guarantees that $y$ and $z$ are distinct.

The latter is a must-not-alias constraint.
Internally, we manipulate a graph of permissions that makes aliasing explicit.

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

Let us see how the type checker represents this type.
A drawing

We can think of \( y \ @ \ xpair \ a \ b \) as the following drawing.
Expanding permissions

Permissions embody **aliasing** relationships

The type-checker first expands \( y \ @ \ xpair \ a \ b \) into

\[
y \ @ \ XPair \{ \text{left: } a; \text{ right: } b \}
\]

then, into

\[
y \ @ \ XPair \{ \text{left: } =l; \text{ right: } =r \}
\]

\[
\ast \ l \ @ \ a
\ast \ r \ @ \ b
\]
The singleton type

=1 is a **singleton type**: \( y \ @ = z \) means \( y \) and \( z \) point to the same object: this is a **must-alias** constraint.
Some syntactic sugar

- We write \( y = z \) for \( y \leftarrow = z \).
- We also write:
  
  \[
  y \leftarrow \text{XPair} \{ \text{left} = l; \text{right} = r \}
  \]
  
  for

  \[
  y \leftarrow \text{XPair} \{ \text{left:} = l; \text{right:} = r \}
  \]
You said dynamic tests

\( y \; @ \; \text{list} \; \tau \) with \( \tau \) exclusive asserts all items in list \( y \) are distinct.

- A mutable, doubly-linked list with arbitrary length,
- A list where an exclusive element is present twice,

... are both situations that cannot be represented statically.
Plan

1. Mezzo pitch
2. A primer on permissions
3. A dynamic discipline of ownership
4. The current state of Mezzo
In a nutshell

We can represent immutable heaps with arbitrary shape, mutable heaps with a tree shape, but we cannot represent mutable heaps with arbitrary shape.

In order to alleviate this restriction, we use dynamic tests to ensure safety. This is achieved through the (new) adoption and abandon operations.
A dynamic discipline of ownership

Our running example

A first-in, first-out queue, and its aliasing pattern.

Cells are mutable; the ownership pattern is no longer a tree.
How does it work? Adoption

An object can be declared as adopting other objects.

exclusive data fifo a =
  | Empty ...
  | NonEmpty ...
adopts cell a

A permission for the adopter (the FIFO) grants permission for its adoptees (the cells).
A dynamic discipline of ownership

An adoption hierarchy

Jonathan Protzenko (INRIA)

The Mezzo language

ML Workshop 2012 25 / 36

f @ fifo

acts as the adopter

c1 @ cell
c2 @ cell

the adoptees of f
How does it work? Adoption (cont'd)

(* x @ cell a * f @ fifo a *)
give x to f;
(* x @ dynamic * f @ fifo a *)

x @ dynamic means “x may currently be adopted by some other object”.

This is a **duplicable** permission.
We traded \texttt{x @ cell a} for \texttt{x @ dynamic}, which is duplicable but \textbf{hides the true type of x}.

\begin{verbatim}
(* x @ dynamic * f @ fifo a *)
take x from f;
(* x @ cell a * f @ fifo a *)
\end{verbatim}

We regain the original permission, but we need to make sure no object can be abandoned twice: \texttt{abandon} involves a \textbf{dynamic check}.
How does it work? Implementation

- Each object contains a hidden field with the address of its adopter, or null.
- The field is set when adopting and cleared when abandoning.
- We perform the check when abandoning an object: its hidden field and the address of (what the user claims is) the adopter must match.
We now explain how the **insert** operation is type-checked.
The FIFO implements the following interface.

```ocaml
type fifo :: TYPE -> TYPE
val create: [a] () -> fifo a
val insert: [a] (consumes a, fifo a) -> ()
val retrieve: [a] fifo a -> option a
```
The insert function

```ocaml
val insert: [a] (consumes a, fifo a) -> ()
let insert [a] (x, f) =
  let c = Cell { data = x; next = () } in
  c.next <- c;
give c to f;
match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
give tail to f;
f.tail <- c
end
```
val insert: [a] (consumes a, fifo a) -> ()
let insert [a] (x, f) =
  let c = Cell { data = x; next = () } in
  c.next <- c;
give c to f;
match f with
| Empty ->
  f <- NonEmpty;
  f.head <- c;
  f.tail <- c
| NonEmpty { tail } ->
  take tail from f;
  tail.next <- c;
give tail to f;
  f.tail <- c
end
val insert: [a] (consumes a, fifo a) -> ()

let insert [a] (x, f) =
  let c = Cell { data = x; next = () } in
  c.next <- c;
  give c to f;
  match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
    give tail to f;
    f.tail <- c
  end
<table>
<thead>
<tr>
<th>Non-duplicable permissions</th>
<th>Duplicable permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>- f @ fifo a</td>
<td>- Ø</td>
</tr>
<tr>
<td>- x @ a</td>
<td></td>
</tr>
<tr>
<td>- c @ Cell { data = x;</td>
<td></td>
</tr>
<tr>
<td>next: () }</td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
let c = Cell { data = x; next = () } in
c.next <- c;
give c to f;
match f with
  | Empty ->
    | f <- NonEmpty;
    | f.head <- c;
    | f.tail <- c
  | NonEmpty { tail } ->
    | take tail from f;
    | tail.next <- c;
    | give tail to f;
    | f.tail <- c
end
```

fifo a) -> ()
### Non-duplicable permissions

- `f @ fifo a`
- `x @ a`
- `c @ Cell { data = x; next = c }`

### Duplicable permissions

- `ø`

```ocaml
let c = Cell { data = x; next = () } in
  c.next <- c;
  give c to f;
match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
    give tail to f;
    f.tail <- c
end
```
val insert: [a] (consumes a, fifo a) -> ()
let insert [a] (x, f) =
  let c = Cell { data = x; next = () } in
  c.next <- c;
  give c to f;
match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
    give tail to f;
    f.tail <- c
end

Non-duplicable permissions
• f @ fifo a
• c @ cell a

Duplicable permissions
• ∅
val insert: [a] (consumes a, fifo a) -> ()
let insert [a] (x, f) =
  let c = Cell { data = x; next = () } in
  c.next <- c;
give c to f;
match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
give tail to f;
  f.tail <- c
end

Non-duplicable permissions
• f @ fifo a

Duplicable permissions
• c @ dynamic
val insert: [a] (consumes a, fifo a) -> ()
let insert [a] (x, f) =
  let c = Cell { data = x; next = () } in
  c.next <- c;
  give c to f;
  match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
    give tail to f;
    f.tail <- c
end
val insert: [a] (consumes a, fifo a) -> ()
let insert [a] (x, f) =
    let c = Cell { data = x; next = () } in
    c.next <- c;
give c to f;
match f with
| Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
| NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
give tail to f;
f.tail <- c
end
val insert: [a] (consumes a, fifo a) -> ()
let insert [a] (x, f) =
  let c = Cell { data = x; next = () } in
  c.next <- c;
give c to f;
match f with
  | Empty ->
      f <- NonEmpty;
      f.head <- c;
      f.tail <- c
  | NonEmpty { tail } ->
      take tail from f;
      tail.next <- c;
give tail to f;
f.tail <- c
end

Non-duplicable permissions
• f @ NonEmpty
  { head = c; tail: () }

Duplicable permissions
• c @ dynamic
val insert: [a] (consumes a, fifo a) -> ()
let insert [a] (x, f) =
  let c = Cell { data = x; next = () } in
c.next <- c;
give c to f;
match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
    give tail to f;
  f.tail <- c
val insert: [a] (consumes a, fifo a) -> ()

let insert [a] (x, f) =
  let c = Cell { data = x; next = () } in
  c.next <- c;
  give c to f;
  match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
    give tail to f;
    f.tail <- c
end

Non-duplicable permissions

• f @ fifo a

Duplicable permissions

• c @ dynamic
val insert: [a] (consumes a, f).
let insert [a] (x, f) =
  let c = Cell { data = x; next = () } in
  c.next <- c;
  give c to f;
match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
    give tail to f;
    f.tail <- c
end

Non-duplicable permissions

• f @ NonEmpty
  { head = head; tail = tail }

Duplicable permissions

• c @ dynamic
• head @ dynamic
• tail @ dynamic
let insert [a] (x, f) =
  let c = Cell { data = x; next = () } in
  c.next <- c;
  give c to f;
  match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
    give tail to f;
    f.tail <- c
end

Non-duplicable permissions

• f @ NonEmpty
  { head = head; tail = tail }

• tail @ cell a

Duplicable permissions

• c @ dynamic
• head @ dynamic

fifo a) -> ()
match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
    give tail to f;
    f.tail <- c
end

Non-duplicable permissions

- f @ NonEmpty
  \{ head = head; tail = tail \}
- tail @ Cell
  \{ data = data; next = next \}
- data @ a

Duplicable permissions

- c @ dynamic
- head @ dynamic
- next @ dynamic
give c to f;
match f with
| Empty ->
  f <- NonEmpty;
  f.head <- c;
  f.tail <- c
| NonEmpty { tail } ->
  take tail from f;
  tail.next <- c;
  give tail to f;
  f.tail <- c
Non-duplicable permissions

- \( f \) @ NonEmpty
  
  \{ head = head; tail = tail \}

- tail @ cell a

Duplicate permissions

- c @ dynamic
- head @ dynamic
- next @ dynamic

let insert \([a]\) \((x, f)\) =

let c = Cell \{ data = x; next = () \} in

c.next <- c;
give c to f;

match f with
| Empty ->
  f <- NonEmpty;
  f.head <- c;
  f.tail <- c
| NonEmpty \{ tail \} ->
  take tail from f;
  tail.next <- c;
  give tail to f;
  f.tail <- c
end
val insert: [a] (consumes a, tail) =>
let insert [a] (x, f) =
  let c = Cell { data = x; next = () } in
  c.next <- c;
give c to f;
m match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
give tail to f;
  f.tail <- c
end

Non-duplicable permissions

- \( f @ \text{NonEmpty} \)
  \{ head = head; tail = tail \}

Duplicable permissions

- \( c @ \text{dynamic} \)
- head @ dynamic
- next @ dynamic
- tail @ dynamic
val insert: [\(a\)] (consumes a, \(\\rightarrow\))

let insert [\(a\)] (x, f) =
  let c = Cell { data = x; next = () } in
  c.next <- c;
  give c to f;
  match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
    give tail to f;
    f.tail <- c

Non-duplicable permissions

- \(f @ \text{NonEmpty}\)
  - \{ head = head; tail = c \}

Duplicable permissions

- \(c @ \text{dynamic}\)
- \(\text{head} @ \text{dynamic}\)
- \(\text{next} @ \text{dynamic}\)
- \(\text{tail} @ \text{dynamic}\)
val insert: [a] (consumes a, consumes a, produces a, produces a)
let insert [a] (x, f) =
  let c = Cell { data = x; next = () } in
  c.next <- c;
  give c to f;
  match f with
  | Empty ->
    f <- NonEmpty;
    f.head <- c;
    f.tail <- c
  | NonEmpty { tail } ->
    take tail from f;
    tail.next <- c;
    give tail to f;
    f.tail <- c
end

Non-duplicable permissions
• f @ fifo a

Duplicable permissions
• c @ dynamic
• head @ dynamic
• next @ dynamic
• tail @ dynamic
Plan

1. Mezzo pitch
2. A primer on permissions
3. A dynamic discipline of ownership
4. The current state of Mezzo
Future work

Concurrency in the style of concurrent separation logic.

Inference e.g. polymorphic function calls.

Proof soundness and type preservation.
The prototype

- We have a **prototype** that successfully type-checks most of the examples found in our tutorial paper (see websites).
- We plan on writing an **interpreter**.
- We started working on better ways to report **error messages**.
Demo time
Thank you
fifo and cell definitions

exclusive data cell a =
  | Cell { data: a; next: dynamic }

exclusive data bag a =
  | Empty { head, tail: () }
  | NonEmpty { head, tail: dynamic; }
adopts cell a
The current state of Mezzo

**length implementation**

```ocaml
val rec length [a] (x: list a): int =
match x with
  | Nil -> 0
  | Cons { tail = tail } -> 1 + length tail
end

val zero = length Nil
```

(this is a real example from the prototype's test suite)
exclusive data xpair a b =
    XPair { left: a; right: b }

val xswap [a, b] (consumes x: xpair a b):
    (| x @ xpair b a) =
    let t = x.left in
    x.left <- x.right;
    x.right <- t

(this is a real example from the prototype's testsuite)
The mode system (1)
Some types are truly affine, e.g.

\[ \text{list (xpair int int)} \]

Some other types are abstract, and must be conservatively treated as affine, such as \( a \) in the body of

\[ \text{length: [a] list a -> int.} \]
The current state of Mezzo

An interface for locks

```plaintext
type lock :: PERM -> TYPE
fact [p :: PERM] duplicable (lock p)
val create: [p :: PERM] () -> lock p
val acquire: [p :: PERM] lock p -> (| p)
val release: [p :: PERM]
  (lock p | consumes p) -> ()
```

The concept of permission plays very nice with locks.
The current state of Mezzo

A bonus feature

```ocaml
data outcome (p :: PERM) =
  | Success { p }
  | Failure {}

val try_acquire: [p :: PERM]
lock p -> outcome p
```

We embed permissions inside a data type definition. When matching on Success, permission p is added to the environment.