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
What is Mezzo? (1)

Mezzo is a strict and impure functional programming language; Mezzo offers a fine-grained control of side-effects, aliasing and ownership.
What is *Mezzo*? (2)

*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.

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

This snippet generates a permission

```ocaml
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 \( y \).
- \texttt{length} requires a permission \( y \ @ \text{list} \ a \) and
- \texttt{return}s the very same permission: this is the default.
- The function also \texttt{produces} a value of type \texttt{int}.
Trading permissions

\texttt{xswap} swaps the two components of a mutable pair.

\texttt{val xswap: [a, b] (consumes y: xpair a b) -> (\mid y \@ xpair b a)}

- we \textit{introduce} \texttt{y} as the name of the argument;
- the argument is \textit{consumed}, i.e. \texttt{y \@ xpair a b} is not returned;
- however, a \textit{new permission} \texttt{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 primer on permissions

A sound example?

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

```ml
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).
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.

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

Let us see how the type checker represents this type.
We can think of \( y \, @ \, \text{xpair} \, a \, b \) as the following drawing.
Permissions embody **aliasing** relationships

The type-checker first **expands** $y \@ xpair\ a\ b$ into

$$y \@ XPair\ \{\ left: a;\ right: b\ \}$$

then, into

$$y \@ XPair\ \{\ left: =l;\ right: =r\ \}$$

$$*\ l\ @\ a$$

$$*\ r\ @\ b$$
=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 \( y \@ =z \).

• We also write:
  \[
  y \@ \text{XPair} \{ \text{left} = l; \text{right} = r \}
  \]
  for
  \[
  y \@ \text{XPair} \{ \text{left:} =l; \text{right:} =r \}
  \]
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).
An adoption hierarchy

A dynamic discipline of ownership

f @ fifo acts as the adopter

c1 @ cell

c2 @ cell

the adoptees of f
A dynamic discipline of ownership

How does it work? Adoption (cont'd)

\[(\ast \ x \ @ \ cell \ a \ \ast \ f \ @ \ fifo \ a \ \ast)\]
give \(x\) to \(f\);
\[(\ast \ x \ @ \ dynamic \ \ast \ f \ @ \ fifo \ a \ \ast)\]

\(x \ @ \ dynamic\) means "\(x\) may currently be adopted by some other object".

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

\[
\begin{align*}
(* x \@ \text{dynamic} * f \@ \text{fifo } a *) \\
take x \text{ from } f; \\
(* x \@ \text{cell } a * f \@ \text{fifo } a *)
\end{align*}
\]

We regain the original permission, but we need to make sure no object can be abandoned twice: abandon involves a 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.
An example!

We now explain how the `insert` operation is type-checked.
The interface for fifos

The FIFO implements the following interface.

```ml
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

```ml
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 @ fifo a
• x @ a

Duplicable permissions
• ø

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
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
• x @ a
• c @ Cell { data = x; next: () }

Duplicate permissions
• Ø
Non-duplicable permissions

- \( f \in \text{fifo} \ a \)
- \( x \in \ a \)
- \( c \in \text{Cell} \{ \text{data} = x; \text{next} = c \} \)

Duplicable permissions

- \( \emptyset \)

```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

• Ø
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

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

Duplicable permissions
• c @ dynamic
Non-duplicable permissions

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

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

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

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

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
Non-duplicable permissions

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

Duplicate permissions

- $c @ \text{dynamic}$
- $\text{head} @ \text{dynamic}$
- $\text{tail} @ \text{dynamic}$

```ocaml
define val insert [a] (consumes a, t.
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
```
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
Non-duplicable permissions

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

Duplicate permissions

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

\[
\text{match } f \text{ with } \\
| \text{Empty } -> \\
  \quad f < - \text{ NonEmpty}; \\
  \quad f.\text{head} < - c; \\
  \quad f.\text{tail} < - c \\
| \text{NonEmpty } \{ \text{tail} \} -> \\
  \quad \text{take} \ \text{tail} \ \text{from} \ f; \\
  \quad \text{tail} . \text{next} < - c; \\
  \quad \text{give} \ \text{tail} \ \text{to} \ f; \\
  \quad f.\text{tail} < - c
\]
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 @ NonEmpty
  { head = head; tail = tail }
- tail @ Cell
  { data = data; next = c }
- data @ a

Duplicable permissions

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

• f @ NonEmpty
  { head = head; tail = tail }
• tail @ Cell
  { data = data; next = c }
• data @ a

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
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
• next @ dynamic
val insert: [a] (consumes a, takes 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
• next @ dynamic
• tail @ dynamic
val insert: [a] (consumes a, tail) = 
  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 = c }

Duplicable permissions

- c @ dynamic 
- head @ dynamic 
- next @ dynamic 
- tail @ 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
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 current state of Mezzo

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
length implementation

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 testsuite)
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 test suite)
The current state of Mezzo

The mode system (1)

- Affine
  - Duplicable
  - Exclusive
Some types are truly affine, e.g.

```
list (xpair int int)
```

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

```
length: [a] list a -> int.
```
The current state of Mezzo

An interface for locks

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