The practice of Mezzo

François Pottier

INRIA

IHP, April 2014
Acknowledgements

Two lectures on Mezzo.

- April 29th, 2pm: motivation and examples.
- April 30th, 4pm: type soundness, data race freedom.
Introduction
- Write-once references: usage
- Mezzo: design principles
- Mezzo: motivation
- Write-once references: interface & implementation

Algebraic data structures

Sharing mutable data

Conclusion
Write-once references: usage
A write-once reference:

- can be written \textit{at most} once;
- can be read only \textit{after} it has been written.

Let us look at a concrete example of use...
open worref
open worref

val r1 = new ()
(* r1 @ writable *)
open worref

val r1 = new ()
(* r1 @ writable *)
val r2 = r1
(* r1 @ writable * r2 = r1 *)
open woref

val r1 = new ()
(* r1 @ writable *)
val r2 = r1
(* r1 @ writable * r2 = r1 *)
val () = set (r1, 3);
(* r1 @ frozen int * r2 = r1 *)
open worref

val r1 = new ()
(* r1 @ writable *)
val r2 = r1
(* r1 @ writable * r2 = r1 *)
val () = set (r1, 3);
(* r1 @ frozen int * r2 = r1 *)
val x2 = get r2
(* r1 @ frozen int * r2 = r1 * x2 @ int *)
open woref

val r1 = new ()
(* r1 @ writable *)
val r2 = r1
(* r1 @ writable * r2 = r1 *)
val () = set (r1, 3);
(* r1 @ frozen int * r2 = r1 *)
val x2 = get r2
(* r1 @ frozen int * r2 = r1 * x2 @ int *)
val rs = (r1, r2)
(* r1 @ frozen int * r2 = r1 * x2 @ int
* rs @ (=r1, =r2) *)
open woref

val r1 = new ()
(* r1 @ writable *)
val r2 = r1
(* r1 @ writable * r2 = r1 *)
val () = set (r1, 3);
(* r1 @ frozen int * r2 = r1 *)
val x2 = get r2
(* r1 @ frozen int * r2 = r1 * x2 @ int *)
val rs = (r1, r2)
(* r1 @ frozen int * r2 = r1 * x2 @ int
 * rs @ (=r1, =r2) *)
(* rs @ (frozen int, frozen int) *)
open worref

val r1 = new ()  (* r1 @ writable *)
val r2 = r1      (* r1 @ writable * r2 = r1 *)
val () = set (r2, 3);  (* r1 @ frozen int * r2 = r1 *)
val x2 = get r2    (* r1 @ frozen int * r2 = r1 * x2 @ int *)
val rs = (r1, r2)  (* r1 @ frozen int * r2 = r1 * x2 @ int *
* rs @ (=r1, =r2) *)
(* rs @ (frozen int, frozen int) *)
Mezzo: design principles
Like a program logic, the static discipline is *flow-sensitive*.

- A *current* (set of) *permission(s)* exists *at each program point*.
- *Different* permissions exist at different points.

Permissions do not exist at runtime.
Thus, there is no such thing as the type of a variable \( x \). Instead,

- at each program point in the scope of \( x \),
- there may be zero, one, or more permissions to use \( x \) in certain ways.
Permissions have *layout* and *ownership* readings.

- e.g., \( r \odot \text{writable} \)

\( x \odot t \) describes the *shape and extent* of a heap fragment, rooted at \( x \), and describes certain *access rights* for it.

“To know about \( x \)” is “to have access to \( x \)” is “to own \( x \)”.
Every permission is either duplicable or affine. At first,

- *Immutable* data is *duplicable*, i.e., shareable.
- *Mutable* data is *affine*, i.e., uniquely owned.
- Mutable data can become immutable; not the converse.
• Writing let \( x = y \) in \( \ldots \) gives rise to an equation \( x = y \).
• It is a permission: \( x @ =y \), where \( =y \) is a singleton type.
• In its presence, \( x @ t \) and \( y @ t \) are interconvertible.
• Thus, any name is as good as any other.
• The same idea applies to let \( x = xs . \text{head} \) in \( \ldots \).
A value can be copied (always). No permission is required.

(* empty *)

\[ \text{let } y = (x, x) \text{ in } \]

(* \(y \) @ \((=x, =x)\) *)
A duplicable permission *can* be copied. This is implicit.

\[
(* \ x \ @ \ \text{int} \ *)
\]

\[
\text{let} \ y = (x, x) \ \text{in}
\]

\[
(* \ x \ @ \ \text{int} \ * \ y \ @ \ (=x, =x) \ *)
\]
A duplicable permission *can* be copied. This is implicit.

\[
\begin{align*}
\ast x @ int & \ast \\
\text{let } y = (x, x) \text{ in} \\
& \ast x @ int \ast y @ (=x, =x) \ast \\
& \ast x @ int \ast y @ (int, int) \ast
\end{align*}
\]
An affine permission \textit{cannot} be copied.

\begin{verbatim}
(* x @ ref int *)
let y = (x, x) in
(* x @ ref int * y @ (=x, =x) *)
\end{verbatim}

In other words, mutable data cannot be shared.
An affine permission cannot be copied.

\[
(* \; x \; @ \; \text{ref int} \; *)
\]

\[
\text{let } \; y \; = \; (x, \; x) \; \text{in}
\]

\[
(* \; x \; @ \; \text{ref int} \; * \; y \; @ \; (=x, \; =x) \; *)
\]

\[
\text{assert } \; y \; @ \; (\text{ref int}, \; \text{ref int}) \; (* \; \text{WRONG!} \; *)
\]

In other words, mutable data cannot be shared.
Examples of duplicable versus affine

• $x \odot \text{list int}$ is duplicable: read access can be shared.
• $x = y$ is duplicable: equalities are forever.
• $x \odot \text{mlist int}$ and $x \odot \text{list (ref int)}$ are affine: they give exclusive access to part of the heap.
Separation

\[ x \ @ \ ref\ int \ \ast\ y \ @ \ ref\ int \ \text{implies} \ x \ \text{and} \ y \ \text{are distinct.} \]

Conjunction is \textit{separating} at mutable data.

\[ z \ @ \ (t,\ u) \ \text{means} \ z \ @ \ (=x,\ =y) \ \ast\ x \ @ \ t \ \ast\ y \ @ \ u, \ \text{for} \ x,\ y \ \text{fresh.} \]

Hence, product is separating.
The same principle applies to records. Hence, list (ref int) denotes a list of distinct references. Mutable data must be tree-structured.

• though x @ ref (=x) can be written and constructed.
Mezzo: motivation
The types of OCaml, Haskell, Java, C#, etc.:

- describe the *structure* of data,
- but do not distinguish *trees* and *graphs*,
- and do not control who has *permission* to read or write.
Could a more ambitious static discipline:

- *rule out* more programming errors,
- and *enable* new programming idioms,
- while remaining reasonably *simple* and *flexible*?
The uniqueness of read/write permissions:

- **rules out**, or helps rule out, several categories of errors:
  - data races;
  - representation exposure;
  - violations of object protocols.

- **allows** the type of an object to vary with time, which enables:
  - explicit memory re-use;
  - gradual initialization;
  - the description of object protocols.
This discipline is restrictive.
Fortunately,

- there is *no restriction* on the use of immutable data;
- there are *several ways* of sharing mutable data:
  - (static) nesting & regions;
  - (dynamic) adoption & abandon;
  - (dynamic) locks.
A few desirable idioms become clumsy or downright impossible.

- e.g., temporarily borrowing an affine element from a container (an array; a region; a user-defined data structure; ...).

Work-arounds: see previous slide.
Write-once references: interface & implementation
A usage protocol can be described in a module signature:

- A *state* is a (user-defined) type.
- A *transition* is a (user-defined) function.
This protocol has two states and four transitions.

```
abstract writable
abstract frozen a
fact duplicable (frozen a)
val new: () -> writable
val set: [a] (consumes r: writable, x: a | duplicable a)
                   -> (| r @ frozen a)
val get: [a] frozen a -> a
```
This protocol has two states and four transitions.

```scala
abstract writable
abstract frozen a
fact duplicable (frozen a)
val new: () -> writable
val set: [a] (consumes r: writable, x: a | duplicable a)
    -> (| r @ frozen a)
val get: [a] frozen a -> a
```
This protocol has two states and four transitions.

abstract writable
abstract frozen a
fact duplicable (frozen a)
val new: () -> writable
val set: [a] (consumes r: writable, x: a | duplicable a)
    -> (| r @ frozen a)
val get: [a] frozen a -> a
This protocol has two states and four transitions.

```
abstract writable
abstract frozen a

fact duplicable (frozen a)

val new: () -> writable

val set: [a] (consumes r: writable, x: a | duplicable a)
    -> (| r @ frozen a)

val get: [a] frozen a -> a
```
This protocol has two states and four transitions.

abstract writable
abstract frozen a
fact duplicable (frozen a)
val new: () -> writable
val set: [a] (consumes r: writable, x: a | duplicable a)
    -> (| r @ frozen a)
val get: [a] frozen a -> a
This protocol has two states and four transitions.

```scala
abstract writable
abstract frozen a
fact duplicable (frozen a)
val new: () -> writable
val set: [a] (consumes r: writable, x: a | duplicable a)
          -> (| r @ frozen a)
val get: [a] frozen a -> a
```

set requires r (dynamic) and r @ writable (static)
This protocol has two states and four transitions.

```scala
abstract writable
abstract frozen a
fact duplicable (frozen a)
val new: () -> writable
val set: [a] (consumes r: writable, x: a | duplicable a) -> (| r @ frozen a)
val get: [a] frozen a -> a
```

consumes keyword means r @ writable NOT returned
This protocol has two states and four transitions.

**abstract** writable

**abstract** frozen a

**fact duplicable** (frozen a)

**val new:** () -> writable

**val set:** [a] (consumes r: writable, x: a | duplicable a)

... -> (| r @ frozen a)

**val get:** [a] frozen a -> a

 duplicable a is a permission
This protocol has two states and four transitions.

abstract writable
abstract frozen a
fact duplicable (frozen a)
val new: () -> writable
val set: [a] (consumes r: writable, x: a | duplicable a)
         -> (| r @ frozen a)
val get: [a] frozen a -> a
This protocol has two states and four transitions.

```plaintext
abstract writable
abstract frozen a
fact duplicable (frozen a)
val new: () -> writable
val set: [a] (consumes r: writable, x: a | duplicable a)
    -> (| r @ frozen a)
val get: [a] frozen a -> a
```

get r requires r @ frozen a
data mutable writable =
    Writable { contents: () }
data frozen a =
    Frozen   { contents: (a | duplicable a) }
val new () : writable =
    Writable { contents = () }
val set [a] (consumes r: writable, x: a | duplicable a) :
       (| r @ frozen a) =
    r.contents <- x;
    tag of r <- Frozen (* this is a no-op *)
val get [a] (r: frozen a) : a =
    r.contents
data mutable writable =  
  Writable { contents: () }

data frozen a =  
  Frozen { contents: (a | duplicable a) }

val new () : writable =  
  Writable { contents = () }

val set [a] (consumes r: writable, x: a | duplicable a)  
  : (| r @ frozen a) =  
  r.contents <- x;  
  tag of r <- Frozen (* this is a no-op *)

val get [a] (r: frozen a) : a =  
  r.contents
data mutable writable =
Writable { contents: () } 
data frozen a =
Frozen { contents: (a | duplicable a) }
val new () : writable =
Writable { contents = () }
val set [a] (consumes r: writable, x: a | duplicable a) :
  (| r @ frozen a) =
r.contents <- x;
tag of r <- Frozen (* this is a no-op *)
val get [a] (r: frozen a) : a =
r.contents
data mutable writable =
  Writable { contents: () }

data frozen a =
  Frozen { contents: {a | duplicable a} }

val new () : writable =
  Writable { contents = () }

val set [a] (consumes r: writable, x: a | duplicable a) :
  (| r @ frozen a) =
r.contents <- x;
tag of r <- Frozen (* this is a no-op *)

val get [a] (r: frozen a) : a =
r.contents
```scala
data mutable writable =
  Writable { contents: () }
data frozen a =
  Frozen { contents: {a | duplicable a} }
val new () : writable =
  Writable { contents = () }
val set [a] (consumes r: writable, x: a | duplicable a) :
  (| r @ frozen a) =
  r.contents <- x;
  tag of r <- Frozen (* this is a no-op *)
val get [a] (r: frozen a) : a =
  r.contents
```

hence,
\[ r @ Writable \{ \text{contents: ()} \} \]
data mutable writable =
    Writable { contents: () }
data frozen a =
    Frozen { contents: (a | duplicable a) }
val new () : writable =
    Writable { contents = () }
val set [a] (consumes r: writable, x: a | duplicable a) :
    (| r @ frozen a) =
    r.contents <- x;
    tag of r <- Frozen (* this is a no-op *)
val get [a] (r: frozen a) : a =
    r.contents
data mutable writable =
  Writable { contents: () }
data frozen a =
  Frozen { contents: (a | duplicable a) }
val new () : writable =
  Writable { contents = () }
val set [a] (consumes r: writable, x: a | duplicable a) : (| r @ frozen a) =
  r.contents <- x;
  tag of r <- Frozen (* this is a no-op *)
val get [a] (r: frozen a) : a =
  r.contents
data mutable writable =
  Writable { contents: () }
data frozen a =
  Frozen { contents: (a | duplicable a) }
val new () : writable =
  Writable { contents = () }
val set [a] (consumes r: writable, x: a | duplicable a)
  : (| r @ frozen a) =
  r.contents <- x;
tag of r <- Frozen (* this is a no-op *)
val get [a] (r: frozen a) : a =
  r.contents
data mutable writable =
    Writable { contents: () }
data frozen a =
    Frozen   { contents: (a | duplicable a) }
val new () : writable =
    Writable { contents = () }
val set [a] (consumes r: writable, x: a | duplicable a) :
    (| r @ frozen a) =
    r.contents <- x;
    tag of r <- Frozen (* this is a no-op *)
val get [a] (r: frozen a) : a =
    r.contents
Outline

- Introduction

- Algebraic data structures
  - Principles
  - Computing the length of a list
  - Melding mutable lists
  - Concatenating immutable lists

- Sharing mutable data

- Conclusion
Principles
The algebraic data type of immutable lists is defined as in ML:

```haskell
data list a =
| Nil
| Cons { head: a; tail: list a }
```
Mutable lists

To define a type of mutable lists, one adds a keyword:

```haskell
data mutable mlist a =
    | MNil
    | MCons { head: a; tail: mlist a }
```
For instance,

- `x @ list int` provides (read) access to an immutable list of integers, rooted at `x`.
- `x @ mlist int` provides (exclusive, read/write) access to a mutable list of integers at `x`.
- `x @ list (ref int)` offers read access to the spine and read/write access to the elements, which are distinct cells.
Permission refinement takes place at case analysis.

```ocaml
match xs with
| MNil ->
  ...
| MCons ->
  let x = xs.head in
  ...
end
```

In contrast, traditional separation logic has *untagged* union.
Permission refinement takes place at case analysis.

```
match xs with
  | MNil ->
  ...
  | MCons ->

  let x = xs.head in

  ...
end
```

A nominal permission:

```
xs @ mlist a
```

In contrast, traditional separation logic has *untagged* union.
Permission refinement takes place at case analysis.

\[
\text{match } \text{xs with}
| \text{MNil} \to
\quad \ldots
| \text{MCons} \to
\quad \text{let } x = \text{xs.head in}
\quad \ldots
\text{end}
\]

In contrast, traditional separation logic has *untagged* union.
Permission refinement takes place at case analysis.

```plaintext
match xs with
  | MNil  ->
    ...
  | MCons ->
    let x = xs.head in
    ...
end
```

Another structural permission:

```
xs @ MCons { head: a; tail: mlist a }
```

In contrast, traditional separation logic has untagged union.
Permission refinement takes place at case analysis.

```ocaml
match xs with
| MNil ->

...|

| MCons ->

let x = xs.head in

...|

end
```

automatically expanded to:

```ocaml
xs @ MCons { head: (=h); tail: (=t) }
* h @ a
* t @ mlist a
```

In contrast, traditional separation logic has *untagged* union.
Permission refinement takes place at case analysis.

```plaintext
match xs with
| MNil  -> or (sugar):
   xs @ MCons { head = h; tail = t }
   * h @ a
   * t @ mlist a

| MCons ->

   let x = xs.head in

   ...

end

In contrast, traditional separation logic has untagged union.
```
Permission refinement takes place at case analysis.

```plaintext
match xs with
| MNil -> ...
| MCons -> let x = xs.head in ...
end
```

In contrast, traditional separation logic has untagged union.

So, after the read access:

```plaintext
xs @ MCons { head = h; tail = t }
* h @ a
* t @ mlist a
* x = h
```
This illustrates two mechanisms:

- A nominal permission can be *unfolded* and *refined*, yielding a structural permission.
- A structural permission can be *decomposed*, yielding separate permissions for the block and its fields.

These reasoning steps are implicit and reversible.
Computing the length of a list
Here is the type of the `length` function for mutable lists.

```ocaml
val length: [a] mlist a -> int
```

It should be understood as follows:

- `length` requires one argument `xs`, along with the permission `xs @ mlist a`.
- `length` returns one result `n`, along with the permission `xs @ mlist a * n @ int`.
val rec length_aux [a] (accu: int, xs: mlist a) : int =
 match xs with
 | MNil ->
   accu
 | MCons ->
   length_aux (accu + 1, xs.tail)
end

val length [a] (xs: mlist a) : int =
 length_aux (0, xs)
val rec length_aux [a] (accu: int, xs: mlist a) : int =
  match xs with
  | MNil ->
    accu
  | MCons ->
    length_aux (accu + 1, xs.tail)
end

val length [a] (xs: mlist a) : int =
  length_aux (0, xs)

initially:
xs @ mlist a
val rec length_aux [a] (accu: int, xs: mlist a) : int =
match xs with
| MNil ->
  accu
| MCons ->
  length_aux (accu + 1, xs.tail)
end

val length [a] (xs: mlist a) : int =
length_aux (0, xs)

upon entry into the first branch:
x @ MNil
val rec length_aux [a] (accu: int, xs: mlist a) : int =
  match xs with
  | MNil ->
    accu
  | MCons ->
    length_aux (accu + 1, xs.tail)
end

val length [a] (xs: mlist a) : int =
  length_aux (0, xs)

upon exit of the first branch:
x @ MNil
val rec length_aux [a] (accu: int, xs: mlist a) : int =
match xs with
  | MNil ->
    accu
  | MCons ->
    length_aux (accu + 1, xs.tail)
end

val length [a] (xs: mlist a) : int =
length_aux (0, xs)
val rec length_aux [a] (accu: int, xs: mlist a) : int =
match xs with
| MNil -> accu
| MCons -> length_aux (accu + 1, xs.tail)
end

val length [a] (xs: mlist a) : int =
length_aux (0, xs)

upon entry into the second branch:
x@MCons { head = h; tail = t }
h@a
t@mlist a
val rec length_aux [a] (accu: int, xs: mlist a) : int =
match xs with
  | MNil ->
    accu
  | MCons ->
    length_aux (accu + 1, xs.tail)
end

val length [a] (xs: mlist a) : int =
length_aux (0, xs)

after the call, nothing has changed:
x @ MCons { head = h; tail = t }
h @ a
t @ mlist a
val rec length_aux [a] (accu: int, xs: mlist a): int =
  match xs with
  | MNil -> accu
  | MCons -> length_aux (accu + 1, xs.tail)
end

val length [a] (xs: mlist a): int =
  length_aux (0, xs)

thus, by recombining:
xs @ MCons { head: a; tail: mlist a }
val rec length_aux [a] (accu: int, xs: mlist a) : int =
match xs with
  | MNil ->
    accu
  | MCons ->
    length_aux (accu + 1, xs.tail)
end

val length [a] (xs: mlist a) : int =
length_aux (0, xs)

thus, by folding:
xs @ mlist a
The analysis of this code is surprisingly simple.

- This is a tail-recursive function, i.e., a loop in disguise.
- As we go, there is a list ahead of us and a list segment behind us.
- Ownership of the latter is implicit, i.e., framed out.

Recursive reasoning, iterative execution.

(Now skipping ahead...
Melding mutable lists
val rec meld_aux [a] (xs: MCons { head: a; tail: mlist a }, consumes ys: mlist a) : () =
match xs.tail with
| MNil ->
  xs.tail <- ys
| MCons ->
meld_aux (xs.tail, ys)
end
val rec meld_aux [a]
  (xs: MCons { head: a; tail: mlist a },
   consumes ys: mlist a) : () =
match xs.tail with
| MNil   ->
   xs.tail <- ys
| MCons  ->
   meld_aux (xs.tail, ys)
end
val rec meld_aux [a]
  (xs: MCons { head: a; tail: mlist a },
   consumes ys: mlist a) : () =
match xs.tail with
| MNil  ->
   xs.tail <- ys
| MCons ->
   meld_aux (xs.tail, ys)
end

at the end, ys is accessible through xs, hence must no longer be used directly.
val rec meld_aux [a]
  (xs: MCons { head: a; tail: mlist a },
   consumes ys: mlist a) : () =
match xs.tail with
| MNil  ->
  xs.tail <- ys
| MCons ->
  meld_aux (xs.tail, ys)
end
val rec meld_aux [a]
  (xs: MCons { head: a; tail: mlist a },
   consumes ys: mlist a) :: () =
match xs.tail with
| MNil  ->
    xs.tail <- ys
| MCons ->
meld_aux (xs.tail, ys)
end
val rec meld_aux [a] (xs: MCons { head: a; tail: mlist a }, consumes ys: mlist a) : () =
match xs.tail with
| MNil ->
  xs.tail <- ys
| MCons ->
meld_aux (xs.tail, ys)
end
val rec meld_aux [a]
(xs: MCons { head: a; tail: mlist a },
 consumes ys: mlist a) : () =
match xs.tail with
| MNil ->
  xs.tail <- ys
| MCons ->
meld_aux (xs.tail, ys)
end
val rec meld_aux [a]
  (xs: MCons { head: a; tail: mlist a },
   consumes ys: mlist a): () =
match xs.tail with
| MNil ->
  xs.tail <- ys
| MCons ->
  meld_aux (xs.tail, ys)
end
val rec meld_aux [a] (xs: MCons { head: a; tail: mlist a }, consumes ys: mlist a) : () =
match xs.tail with
| MNil ->
  xs.tail <- ys
| MCons ->
meld_aux (xs.tail, ys)
end
val rec meld_aux [a]
  (xs: MCons { head: a; tail: mlist a },
   consumes ys: mlist a) : () =
  match xs.tail with
  | MNil  ->
      xs.tail <- ys
  | MCons ->
      meld_aux (xs.tail, ys)
end

xs @ MCons { head: a; tail = t }
t @ mlist a
val rec meld_aux [a]
  (xs: MCons { head: a; tail: mlist a },
   consumes ys: mlist a) : () =
match xs.tail with
| MNil  ->
  xs.tail <- ys
| MCons ->
  meld_aux (xs.tail, ys)
end
val meld [a] (consumes xs: mlist a, consumes ys: mlist a) : mlist a =

match xs with
| MNil -> ys
| MCons -> meld_aux (xs, ys); xs
end
Concatenating immutable lists
Three states

An \texttt{MCons} cell:
- mutable,
- uninitialized \texttt{tail},
- type: \texttt{MCons} \{ head: a; tail: () \}

An isolated \texttt{Cons} cell:
- immutable,
- \textit{not} the start of a well-formed list,
- type: \texttt{Cons} \{ head: a; tail = t \}

A list cell:
- immutable,
- the start of a well-formed list,
- type \texttt{list a}
The big picture
The big picture

MCons
head
tail

MCons
head
tail

Cons
head
tail

Cons
head
tail

Cons
head
tail

Cons
head
tail

Cons
head
tail

Cons
head
tail

xs

ys
The big picture

xs

Cons
head
tail

Cons
head
tail

Cons
head
tail

Cons
head
tail

Cons
head
tail

ys

Cons
head
tail

Cons
head
tail

Cons
head
tail

Cons
head
tail
The big picture
The big picture

Cons head tail
Cons head tail
MCons head tail
Cons head tail
Cons head tail
Cons head tail
Cons head tail
(xs)

Cons head tail
Cons head tail
Cons head tail
Cons head tail
Cons head tail
Cons head tail
Cons head tail
(ys)
The big picture

(xs -> Cons head tail) -> Cons head tail -> Cons head tail -> Cons head tail

(xs) -> Cons head tail -> Cons head tail -> Cons head tail

(ys) -> Cons head tail -> Cons head tail

(xs) -> Cons head tail -> Cons head tail

(ys)
The big picture

```
xs

Cons
head
tail

ys

Cons
head
tail

Cons
head
tail

Cons
head
tail

Cons
head
tail

Cons
head
tail

Cons
head
tail
```
The big picture

[Diagram showing a recursive structure with 'Cons' nodes connected by arrows labeled 'xs' and 'ys'.]
The big picture

```
Cons
  head
  tail

Cons
  head
  tail

Cons
  head
  tail

Cons
  head
  tail
```

xs

ys
val rec append_aux [a] (consumes (  
  dst: MCons { head: a; tail: () },  
  xs: list a, ys: list a  
)) : (| dst @ list a) =  
  match xs with  
  | Cons ->  
    let dst' = MCons { head = xs.head; tail = () } in  
    dst.tail <- dst';  
    tag of dst <- Cons;  
    append_aux (dst', xs.tail, ys)  
  | Nil ->  
    dst.tail <- ys;  
    tag of dst <- Cons  
  end
val rec append_aux [a] (consumes (dst: MCons { head: a; tail: () }, xs: list a, ys: list a)): (dst @ list a) = match xs with
| Cons ->
  let dst' = MCons { head = xs.head; tail = () } in
dst.tail <- dst';
tag of dst <- Cons;
append_aux (dst', xs.tail, ys)
| Nil ->
dst.tail <- ys;
tag of dst <- Cons
end

all three inputs are consumed
val rec append_aux [a] (consumes (  
dst: MCons { head: a; tail: () },  
xs: list a, ys: list a  
)) : (| dst @ list a) =  
match xs with  
  | Cons ->  
    let dst' = MCons { head = xs.head; tail = () } in  
    dst.tail <- dst';  
    tag of dst <- Cons;  
    append_aux (dst', xs.tail, ys)  
  | Nil ->  
    dst.tail <- ys;  
    tag of dst <- Cons  
end  

dst is initially unfinished
val rec append_aux [a] (consumes (
    dst: MCons { head: a; tail: () },
    xs: list a, ys: list a
)) : (| dst @ list a) =
match xs with
| Cons ->
  let dst' = MCons { head = xs.head; tail = () } in
  dst.tail <- dst';
  tag of dst <- Cons;
  append_aux (dst', xs.tail, ys)
| Nil ->
  dst.tail <- ys;
  tag of dst <- Cons
end

xs and ys are initially valid
val rec append_aux [a] (consumes (  
dst: MCons { head: a; tail: () },  
xs: list a, ys: list a) : (| dst @ list a) -  
match xs with  
| Cons ->  
  let dst' = MCons { head = xs.head; tail = () } in  
dst.tail <- dst';  
tag of dst <- Cons;  
append_aux (dst', xs.tail, ys)  
| Nil ->  
dst.tail <- ys;  
tag of dst <- Cons  
end

upon return, dst is valid
val rec append_aux [a] (consumes (  
dst: MCons { head: a; tail: () },  
xs: list a, ys: list a  
)) : (| dst @ list a) =  
match xs with  
| Cons ->  
  let dst' = MCons { head = xs.head; tail = () } in  
  dst.tail <- dst';  
  tag of dst <- Cons;  
  append_aux (dst', xs.tail, ys)  
| Nil ->  
  dst.tail <- ys;  
  tag of dst <- Cons  
end
val rec append_aux [a] (consumes (  
  dst: MCons { head: a; tail: () },  
  xs: list a, ys: list a  
)) : (| dst @ list a) =  
match xs with  
| Cons ->  
  let dst' = MCons { head = xs.head; tail = () } in  
  dst.tail <- dst';  
  tag of dst <- Cons;  
  append_aux (dst', xs.tail, ys)  
| Nil ->  
  dst.tail <- ys;  
  tag of dst <- Cons  
end
val rec append_aux [a] (consumes (  
dst: MCons { head: a; tail: () },  
xs: list a, ys: list a)) : (| dst @ list a, xs @ Cons { head = h; tail = t }  
dst @ Cons { head: a; tail = dst' }  
dst' @ MCons { head: a; tail: () }  
t @ list a  
ys @ list a)  
match xs with  
| Cons ->  
  let dst' =  
    dst @ Cons { head: a; tail = dst' }  
  in  
  dst.tail <- dst';  
tag of dst <- Cons;  
append_aux (dst', xs.tail, ys)  
| Nil ->  
  dst.tail <- ys;  
tag of dst <- Cons  
end
val rec append_aux [a] (consumes ( 
    dst: MCons { head: a; tail: () },
    xs: list a, ys: list a )) : (| dst @ list a |
match xs with
| Cons ->
    let dst' = MCons { head: xs.head; tail: dst.tail },
    tag of dst <- Cons;
    append_aux (dst', xs.tail, ys)
| Nil ->
    dst.tail <- ys;
    tag of dst <- Cons
end
val rec append_aux [a] (consumes (dst: MCons { head: a; tail: () }), xs: list a, ys: list a)): ([ ] dst @ list a) = match xs with | Cons -> let dst' = dst @ Cons { head: a; tail = dst' } in dst.tail <- dst'; tag of dst <- Cons; append_aux (dst', xs.tail, ys) | Nil -> dst.tail <- ys; tag of dst <- Cons end
val rec append_aux [a] (consumes (  
dst: MCons { head: a; tail: () },
xs: list a, ys: list a)) : (| dst @ list a |
match xs with  
| Cons ->  
  let dst' = ...
  dst.tail <- dst',
  tag of dst <- Cons;
  append_aux (dst', xs.tail, ys)
| Nil ->  
  dst.tail <- ys;
  tag of dst <- Cons
end

val rec append_aux [a] (consumes (  
dst: MCons { head: a; tail: () },  
xs: list a, ys: list a)) : (| dst @ list a)  
match xs with  
  | Cons ->  
    let dst' = MCons { head: xs, tail: () },  
    dst.tail <- dst',  
    tag of dst <- Cons;  
    append_aux (dst', xs.tail, ys)  
  | Nil ->  
    dst.tail <- ys;  
    tag of dst <- Cons  
end
val append [a] (consumes (xs: list a, ys: list a)) : list a =
  match xs with
  | Cons ->
      let dst = MCons { head = xs.head; tail = () } in
      append_aux (dst, xs.tail, ys);
      dst
  | Nil ->
      ys
end
Remark

The type of append:

\[[a] \text{(consumes (list a, list a))} \rightarrow \text{list a}\]

is a subtype of:

\[[a] \text{(list a, list a | duplicable a)} \rightarrow \text{list a}\]

The arguments are consumed \textit{only if not duplicable}. 
Introduction

Algebraic data structures

Sharing mutable data
  - Nesting and regions
  - Adoption and abandon
  - Locks

Conclusion
Nesting and regions


*Nesting* (Boyland, 2010) is a static mechanism for organizing permissions into a hierarchy. Conceptually, the hierarchy is constructed as the program runs. Nesting is *monotonic*: the hierarchy grows with time.
Nesting can be *axiomatized* in Mezzo.
This extension has not been proven sound. It could be (I think).
Details omitted.
Static *regions* can be *defined* on top of nesting.
An affine type of regions - internally defined as the unit type:

```ocaml
abstract region
val newregion: () -> region
```

A *duplicable* type of references that inhabit a region:

```ocaml
abstract rref (rho : value) a
fact duplicable (rref rho a)
```

These references can be shared without restriction.
val newrref: (consumes x: a | rho @ region) -> rref rho a
val get: (r: rref rho a | duplicable a | rho @ region) -> a
val set: (r: rref rho a, consumes x: a | rho @ region) -> ()

All three are polymorphic in rho and a. Quantifiers omitted. The token rho @ region is required to use any reference in rho. The references are collectively “owned by the region”. This subsumes Haskell's ST monad.
Nesting and regions have *no runtime cost*. However,

- `get` must be *restricted to duplicable elements* (prev. slide).
- Handling affine elements requires a more clumsy mechanism for *focusing on at most one element* at a time. 
  - Focusing on two elements would entail a proof obligation: \( x \neq y \).
- Membership in a region *cannot* be revoked.
Adoption and abandon
What if something like regions existed \textit{at runtime}?
Old idea, if one thinks of a region as a “memory allocation area”. Here, however, there is a single garbage-collected heap. We are thinking of a “region” as a “unit of ownership”.
Imagine a “region” is a runtime object that maintains a list of its “members”.

We prefer to speak of adopter and adoptees.

Conceptually,

- **Adoption** adds an adoptee to the list.
- **Abandon** takes an adoptee out of the list,
  - after checking *at runtime* that it is there!
This removes the difficulties with static regions.

- an adopter-adoptee relationship *can* be revoked.
- “focusing” amounts to *taking* an adoptee away from its adopter, then *giving* it back.
- “focusing” on multiple elements is permitted.
  - they must be distinct, or the program *fails* at runtime!
Searching a linked list of adoptees would be too slow. Instead, each adoptee points to its adopter (if it has one). Every object has a special adopter field, which may be null.

- Adoption, `give x to y`, means:
  \[x.\text{adopter} \leftarrow y\]
- Abandon, `take x from y`, means:
  \[
  \begin{aligned}
  &\text{if } x.\text{adopter} == y \\
  &\text{then } x.\text{adopter} \leftarrow \text{null} \\
  &\text{else fail}
  \end{aligned}
  \]
An adopter *owns* its adoptees.

Adoption and abandon are very much like *inserting* and *extracting* an element out of a *container*:

- both require a permission for the adopter;
- adoption *consumes* a permission for the new adoptee; abandon allows *recovering* it.
An adopter *owns* its adoptees. Adoption and abandon are very much like *inserting* and *extracting* an element out of a *container*:

- both require a permission for the adopter;
- adoption *consumes* a permission for the new adoptee; abandon allows *recovering* it.

Demo!
Sharing mutable data

Locks
Regions and adoption-and-abandon serve a common purpose:

• move from one-token-per-object to `one-token-per-group`;
• introduce a `duplicable` type of pointer-into-the-group;
• thus permitting `aliasing` within a group.
A problem remains, though:

- every bit of mutable state is controlled by *some* unique token;
- i.e., every side effect *must* be advertised in a function's type;
- thus, multiple clients *must* coordinate and exchange a token.

There is a certain lack of modularity.
Consider a “counter” abstraction, encapsulated as a function.

- it has *abstract* state: its type is \{p : perm\} ((| p) -> int | p).
- it *cannot* be shared by two threads,
  - unless they *synchronize* and exchange p;
  - without synchronization, there would be a *data race*!

A well-typed Mezzo program is data-race free.
Consider a “counter” abstraction, encapsulated as a function.

- it has *abstract* state: its type is \( \{ p : \text{perm} \} (\| p \| \rightarrow \text{int} \mid p) \).
- it *cannot* be shared by more threads,
  - unless they *synchronize* to exchange \( p \);
  - without synchronization, there would be a *data race*!

A well-typed Mezzo program is data-race free.
Introducing a *lock* at the same time:

- removes the data race,
- allows the counter to have type `()` -> `int`.

The counter now has *hidden state*.

Let's see how this works...
The axiomatization of locks begins with two abstract types:

```plaintext
abstract lock (p: perm)
fact duplicable (lock p)
```

```plaintext
abstract locked
```

The permission \( p \) is the *lock invariant*. 
The basic operations are:

```
val new:
    (| consumes p)   -> lock p

val acquire:
    (l: lock p)     -> (| p * l @ locked)

val release:
    (l: lock p | consumes (p * l @ locked)) -> ()
```

All three are polymorphic in p. Quantifiers omitted.
While the lock is unlocked, one can think of \( p \) as **owned by the lock**. The lock is **shareable**, since \( \text{lock } p \) is duplicable. Hence, a lock allows **sharing** and **hiding** mutable state.
The pattern of *hiding* a function's internal state can be encoded once and for all as a second-order function:

```haskell
val hide : [a, b, p : perm] (f : (a | p) -> b | consumes p) -> (a -> b)
```
Hiding as a design pattern

```haskell
val hide [a, b, p : perm] (f : (a | p) -> b | consumes p) : (a -> b) =
  let l : lock p = new () in
  fun (x : a) : b =
    acquire l;
    let y = f x in
    release l;
    y
```
Hiding as a design pattern

```plaintext
val hide [a, b, p : perm] (f : (a | p) -> b | consumes p) : (a -> b) =
  let l : lock p = new () in
  fun (x : a) : b =
    acquire l;
    let y = f x in
    release l;
    y
```
The pattern of hiding a function's internal state can be encoded once and for all as a second-order function:

```plaintext
val hide [a, b, p : perm] (f : (a | p) -> b | consumes p) : (a -> b) =
  let l : lock p = new () in
  fun (x : a) : b =
    acquire l;
    let y = f x in
    release l;
    y
```

`l @ lock p` because it is duplicable.
Hiding as a design pattern

```haskell
val hide [a, b, p : perm] (f : (a | p) -> b | consumes p) : (a -> b) =
  let l : lock p = new () in
  fun (x : a) : b =
    acquire l;
    let y = f x in
    release l;
    y
```

l @ lock p
l @ locked
p
Hiding as a design pattern

```plaintext
val hide [a, b, p : perm] (f : (a | p) -> b | consumes p) : (a -> b) =
  let l : lock p = new () in
  fun (x : a) : b =
    acquire l;
    let y = f x in
    release l;
    y
```
Hiding as a design pattern

val hide [a, b, p : perm] (f : (a | p) -> b | consumes p) : (a -> b) =
  let l : lock p = new () in
  fun (x : a) : b =
    acquire l;
    let y = f x in
    release l;
    y
Rules of thumb

Regarding *regions* versus *adoption and abandon*,

- they serve the same purpose, namely *one-token-per-group*;
- use regions if possible, otherwise adoption and abandon.

Regarding *locks*,

- they serve a different purpose, namely *no-token-at-all*;
- they are typically used *in conjunction* with the above.
  - a lock protects a token that controls a group of objects.
Outline

- Introduction
- Algebraic data structures
- Sharing mutable data
- Conclusion
Sources of inspiration

Mezzo draws inspiration from many sources. Most influential:

- **Linear and affine types** (Wadler, 1990) (Plasmeijer et al., 1992).
  - not every value can be copied!

- **Alias types** (Smith, Walker & Morrisett, 2000), $L^3$ (Ahmed, Fluet & Morrisett 2007).
  - copying a value is harmless,
  - but not every capability can be copied!
  - keep track of equations between values via singleton types.

- Regions and focusing in **Vault** (Fähndrich & DeLine, 2002);

- **Separation logic** (Reynolds, 2002) (O'Hearn, 2007).
  - ownership is in the eye of the beholder.
  - separation by default; local reasoning.
  - a lock owns its invariant.
What distinguishes Mezzo?

A *high-level* underlying untyped programming language:

- algebraic data types preferred to records and null pointers;
- (tail) recursion preferred to iteration;
- garbage collection, first-class functions, etc.
What distinguishes Mezzo?

A *conceptual framework* that helps structure programs.

- should help design more reliable programs;
- could help carry out proofs of programs.
At the present time I think we are on the verge of discovering at last what programming languages should really be like. [...] My dream is that by 1984 we will see a consensus developing for a really good programming language [...]
At the present time I think we are on the verge of discovering at last what programming languages should really be like. [...] My dream is that by 1984 we will see a consensus developing for a really good programming language [...] 

Donald E. Knuth, 1974.
What distinguishes Mezzo?

Technically, some novel features of Mezzo are:

• the permission discipline *replaces* the type discipline;
• a new view of algebraic data types, with nominal and structural permissions, and a new “tag update” instruction;
• a new, lightweight treatment of the distinction between duplicable and affine data;
• adoption and abandon.
The project was launched in late 2011 and has involved

- Jonathan Protzenko (Ph.D student, soon to graduate),
- Thibaut Balabonski (post-doc researcher),
- Henri Chataing, Armaël Guéneau, Cyprien Mangin (interns),
- and myself (INRIA researcher).
We currently have:

- a *type soundness proof* for a subset of Mezzo (next lecture!);
- a working *type-checker*;
- a “compiler” down to untyped OCaml.
Many questions!

- Can we improve *type inference* and type error reports?
- Is this *a good mix* between static and dynamic mechanisms?
- What about temporary *read-only views* of mutable objects?
- Can we express complex *object protocols*?
- What about specifications & *proofs* of programs?
More information online:
http://gallium.inria.fr/~protzenk/mezzo-lang/