An overview of Mezzo

François Pottier

INRIA

Bertinoro, June 2015
Acknowledgements

What is Mezzo?

An experimental programming language in the tradition of ML.
Try it out in your browser:

http://gallium.inria.fr/~protzenk/mezzo-web/

Or install it:

opam install mezzo
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, including *data races*,
- and *enable* new programming idioms,
- while remaining reasonably *simple* and *flexible*?
A quick comparison

In comparison with Tobias Wrigstad's talk (yesterday),

- *data race freedom* and *ownership transfer* are goals too;
- getting rid of GC is not;
- types and permissions *do not* influence code generation; they are erased at runtime.
A first example and a few principles
- Write-once references: usage
- Mezzo: (some) design principles
- Write-once references: interface & implementation
- Mezzo: the good and the bad

Algebraic data structures

Sharing mutable data

Conclusion
A first example and a few principles

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 woref
open woref

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

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

```ocaml
val r1 = new ()
(* r1 @ writable *)
val r2 = r1
(* r1 @ writable * r2 = r1 *)
val () = set (r1, 3);
(* r1 @ frozen int * r2 = r1 *)
```
### Usage

- **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 *)
  ```
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 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) *)
Mezzo: (some) 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 \textit{the} type of a variable $x$. Instead,

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

- e.g., \( r @ w \) writable

\( x @ t \) 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.

The basic rules are:

- **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 ...` 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.head in ...`.
A value can be copied (always). No permission is required.

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

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

\[
(* x \at\ int *)
\]

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

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

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

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

```
let y = (x, x) in
(\* x @ ref int \* y @ (=x, =x) \*)
```
An affine permission cannot be copied.

(* x @ ref int *)

let y = (x, x) in

(* x @ ref int * y @ (=x, =x) *)

assert y @ (ref int, ref int) (* WRONG! *)

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

- $x \mathrel{@} \text{list int}$ is duplicable: read access can be shared.
- $x = y$ is duplicable: equalities are forever.
- $x \mathrel{@} \text{mlist int}$ and $x \mathrel{@} \text{list (ref int)}$ are affine: they give exclusive access to part of the heap.
\[ x \in \text{ref int} \times y \in \text{ref int} \text{ implies } x \text{ and } y \text{ are distinct.} \]
Conjunction is \textit{separating} at mutable data.
\[ z \in (t, u) \text{ means } z \in (\leftarrow x, \leftarrow y) \times x \in t \times y \in u, \text{ for } x, y \text{ fresh.} \]
Hence, product is separating.
The same principle applies to records. Hence, \texttt{list (ref int)} denotes a list of \textit{distinct} references.Mutable data must be \textit{tree}-structured.

- though \texttt{x @ ref (=x)} can be written and constructed.
A first example and a few principles

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.
Specification of write-once refs

This protocol has two states and four transitions.
This is the interface file \texttt{woref.mzi}:

```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
```
This protocol has two states and four transitions.

This is the interface file `woref.mzi`:

```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
```
This protocol has two states and four transitions.

This is the interface file `woref.mzi`:

```
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.

This is the interface file `woref.mzi`:

```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
```

Implicit transition from `frozen` to `frozen * frozen`
This protocol has two states and four transitions.

This is the interface file `woref.mzi`:

```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
```

Explicit transition into writable
This protocol has two states and four transitions.

This is the interface file `woref.mzi`:

```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
```

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

This is the interface file `woref.mzi`:

```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
```

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

This is the interface file `woref.mzi`:

```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
```

duplicable a is a permission

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

This is the interface file `woref.mzi`:

```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
```

explicit transition from writable to frozen
This protocol has two states and four transitions.

This is the interface file `woref.mzi`:

```ocaml
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
```
This is the implementation file `woref.mz`:

```haskell
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
```
This is the implementation file woref.mz:

```haskell
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
```

a field of type ()
This is the implementation:

```haskell
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
```

a field of type a where a must be duplicable

A field of type `a` where `a` must be duplicable.
This is the implementation:

```haskell
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
```
This is the implementation:

```haskell
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, 

\[ r \circledast \text{Writable} \{ \text{contents: ()} \} \]
This is the implementation:

```hs
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
```
This is the implementation:

```haskell
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,

```haskell
  r @ Writable { contents: a }
```
This is the implementation:

```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
```

after the tag update, `r @ Frozen { contents: a }`
This is the implementation:

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
A first example and a few principles

Mezzo: the good and the bad
The uniqueness of read/write permissions:

- *rules out* several categories of errors:
  - data races; hence, *shared-memory concurrency is safe*;
  - representation exposure;
  - violations of (certain) object protocols.

- *allows* the type of an object to vary with time, which enables:
  - explicit memory re-use;
  - gradual initialization;
  - describing (certain) object protocols.
Here are some other positive aspects:

- all of the *power* of ML, and more;
  - higher-order functions, pattern matching, polymorphism, etc.
- no need to annotate types with owners;
  - to have a permission is to own
- *ownership transfer* is easy;
  - just pass (or return, or store, or extract) a permission
- no need to annotate function types with effects.
  - just pass and return a permission
Moving an element *into* or *out of* a container is easy.
Here is a typical container interface:

```scala
defabstract bag a
def val new: [a] () -> bag a
def val insert: [a] (bag a, consumes a) -> ()
def val extract: [a] bag a -> option a
```
The discipline *forbids sharing* mutable data.

For this reason, *borrowing* an element from a container is typically restricted to *duplicable* elements:

```haskell
val find:
  [a]
  duplicable a =>
  (a -> bool) -> list a -> option a
```

This affects user-defined data structures, arrays, regions, etc.
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 first example and a few principles

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

Sharing mutable data

Conclusion
(More) Principles
The algebraic data type of immutable lists is defined as in ML:

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

```
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.

```haskell
match xs with
| MNil ->

... 

| MCons ->

let x = xs.head in

...

dend
```

A nominal permission:

```
xs @ mlist a
```

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
```

A structural permission:

```
xs @ MNil
```

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.

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

automatically expanded to:

```
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.

```ml
match xs with
| MNil ->

... 

| MCons ->

let x = xs.head in

...

end
```

or (sugar):

```ml
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 ->
  ...,
  | MCons ->

  let x = xs.head in
  ...
end
```

so, after the read access:

```plaintext
xs @ MCons { head = h; tail = t }
* h @ a
* t @ mlist a
* x = h
```

In contrast, traditional separation logic has untagged union.
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
Implementation

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:
xs @ 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:
xs @ 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 @ 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 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:
xs @ 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.
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

xs is not consumed: at the end,
it is still a valid non-empty list
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

xs @ MCons { head: a; tail = ys }
t @ MNil
ys @ 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 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
Melding mutable lists (1/2)

```ocaml
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 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 `MCons` cell:
- mutable,
- uninitialized tail,
- type: `MCons { head: a; tail: () }`

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

A list cell:
- immutable,
- the start of a well-formed list,
- type `list a`
The big picture
The big picture
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

ys
The big picture

xs

ys
The big picture
The big picture

```
Cons head tail
Cons head tail
Cons head tail
Cons 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
Cons head tail

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 (d underside the contents of the image, ensuring the text is clear and readable.)

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  
  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) : | dst @ list a)
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 <- 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 @ 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 @ Cons { head: xs.1, tail: xs.2 },
    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 append : list 'a -> (list 'a * 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:

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

is a subtype of:

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

The arguments are consumed \textit{only if not duplicable}. 
A first example and a few principles

Algebraic data structures

Sharing mutable data
  - Regions (and nesting)
  - Adoption and abandon
  - Locks

Conclusion
An affine permission is a (static) unique token. We have seen that we can

• aggregate several tokens, yielding a token for a (tree-structured) composite object
• conversely, split a token for a tree into separate tokens for the root and sub-trees
We have seen that *pointer* and *permission* are distinct concepts: either one can exist without the other.

We have exploited this *at a very local scale*, e.g. when type-checking `meld` and `append`.

Yet, we have *not* exploited this in algebraic data type definitions.

- we always marry a pointer to a sub-tree and a permission to access it
As long as we stick to this style, we cannot express:

- *aliasing*, where an object is accessible via two pointers;
- *shared memory*, where an object is accessible to two threads.
What do we need?

We need ways of saying, roughly,

- “this is a pointer...”
- “...without a permission...”
- “...but here is how to get the permission when needed.”
Sharing mutable data

Regions (and nesting)
A region is a *group* of objects (of identical type). There is *one permission for the group*, instead of one per object. A region does not exist at runtime. It is imaginary. See e.g. Haskell's ST monad. See also Cyclone (Swamy et al., 2006).
An affine type of regions - internally defined as the unit type:

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

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

```-scala
abstract rref (r : value) a
fact duplicable (rref r a)
```

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

All three are polymorphic in r and a. Quantifiers omitted.
The token r @ region is required to use any reference in r.
The references are collectively “owned by the region”.
Regions have *no runtime cost*. However,

- `get` is *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, also known as *multi-focusing*, would entail a proof obligation: $x \neq y$.
- Membership in a region *cannot* be revoked.
Nesting (Boyland, 2010) is a static mechanism for organizing permissions into a hierarchy.
The hierarchy is constructed as the program runs and grows with time.
Nesting can be axiomatized in Mezzo (by adding a few primitive operations which do nothing at runtime).
Regions can be defined as a library on top of nesting.
Like regions, nesting has limitations (prev. slide).
Adoption and abandon
What if something like regions existed *at runtime*?

Old idea, if one thinks of a region as a “memory allocation area”.

- Tofte and Talpin, 1994

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** (a.k.a. give) adds an adoptee to the list.
- **Abandon** (a.k.a. take) extracts an adoptee from the list,
  - and fails *at runtime* if it isn't in the list!
Adoption and abandon

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!
A FIFO queue as a linked list with `first` and `last` pointers. There is *aliasing*. This cannot be type-checked in vanilla Mezzo. We let the “queue” object adopt all of the “list cell” objects. The code type-checks (but could fail at runtime if we mistakenly break our intended invariant).

See P. and Protzenko, ICFP 2013.
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:
  $x$.adopter <- $y$

- Abandon, take $x$ from $y$, means:
  
  if $x$.adopter == $y$
  then $x$.adopter <- null
  else fail
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!
Locks
Towards hidden state

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

- move from one-token-per-object to \textit{one-token-per-group};
- introduce a \textit{duplicable} type of pointer-into-the-group;
- thus permitting \textit{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 \textit{abstract} state: its type is \{p : \texttt{perm}\} ((| p) -> \texttt{int} | p).
- it \textit{cannot} be shared by two threads,
  - unless they \textit{synchronize} and exchange \texttt{p};
  - without synchronization, there would be a \textit{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 \(\{p : \text{perm}\} \ (| \ p \ | \rightarrow \text{int} \ | \ p\)).
- 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.
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:

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

abstract locked

The permission p is the lock invariant.
The basic operations are:

**val new:**

\[ (\mid \text{consumes } p) \rightarrow \text{lock } p \]

**val acquire:**

\[ (l: \text{lock } p) \rightarrow (\mid p \ast l @ \text{locked}) \]

**val release:**

\[ (l: \text{lock } p \mid \text{consumes } (p \ast l @ \text{locked})) \rightarrow () \]

All three are polymorphic in \( p \). Quantifiers omitted.
From concurrent separation logic (O'Hearn, 2007). 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:

```scala
val hide : [a, b, p : perm] (    f : (a | p) -> b    | consumes p ) -> (a -> b)
```
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
```

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
```

l @ lock p
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

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

```ocaml
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
```

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

```ocaml
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
```
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.
A first example and a few principles

Algebraic data structures

Sharing mutable data

Conclusion
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?

It is a *high-level* programming language:

- algebraic data types preferred to records and null pointers;
- (tail) recursion preferred to iteration;
- garbage collection, first-class functions, polymorphism, etc.
- to some extent, lightweight types (i.e., no owner annotations).
Shortcomings

It is far from perfect:

- type inference can be unpredictable;
- it takes a black belt to understand type errors;
- there is currently no interoperability with OCaml.
At the present time I think we are on the verge of discovering at last what programming languages should really be like. [...]
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.
More information online:
http://gallium.inria.fr/~protzenk/mezzo-lang/
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).
Where we are

We currently have:

- a type soundness proof for a subset of Mezzo;
- 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?