A bird's eye view of Mezzo

François Pottier  Jonathan Protzenko

INRIA

MSR/INRIA, Nov 2012
What for?

How?

A tiny taste

Project status
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*?
We would like to *rule out*:

- representation exposure;
- data races;
- violations of object protocols;

and to *enable*:

- gradual initialization;
- (in certain cases) explicit memory re-use.
Outline

- What for?
- How?
- A tiny taste
- Project status
A variable $x$ does not have a fixed type throughout its lifetime. Instead,

- *at each point* in the scope of $x$,
- one may or may not have *permission* to use $x$ in certain ways.
The system imposes a global invariant: at any time,

- if $x$ is a mutable object, there exists \textit{at most one} permission to read and write $x$;
- if $x$ is an immutable object, there may exist arbitrarily \textit{many} permissions to read $x$. 
Why is this a useful discipline?

The uniqueness of read/write permissions:

• *rules out* representation exposure and data races;
• *allows* the type of an object to vary with time, which enables the enforcement of object protocols, gradual initialization, etc.
Isn't this a restrictive discipline?

Yes, it is, but:

• there is *no restriction* on the use of immutable data;
• there is an *escape hatch* that involves dynamic checks.
What for?

How?

A tiny taste

Project status
Concatenating two *immutable* lists creates sharing:

```
let xs : list int = ... in
let ys : list int = ... in
let zs : list int = concat(xs, ys) in
...
```

The lists $xs$ and $zs$ have common elements.
The lists $ys$ and $zs$ have common elements and cells.
This is harmless. We would like to *accept* this code.
What if the lists have \textit{mutable} elements?

```ml
let xs : list (ref int) = ... in
let ys : list (ref int) = ... in
let zs : list (ref int) = concat(xs, ys) in
...
```

Some elements are accessible via $xs$ and $zs$, or via $ys$ and $zs$. This is potentially dangerous.

We would like to \textit{accept} this code yet \textit{prevent} the programmer from using (say) $xs$ and $zs$ as if they were physically disjoint.
In *Mezzo*, the first code snippet gives rise to three permissions:

```
x @ list int
ys @ list int
zs @ list int
```

All three lists can be freely used in the code that follows.
Reasoning with permissions

The first two lines of the second code snippet give rise to:

\[
\begin{align*}
xs & \in \text{list (ref int)} \\
y s & \in \text{list (ref int)}
\end{align*}
\]

These permissions are *consumed* at line three, which gives rise to:

\[
zs \in \text{list (ref int)}
\]

At the end, \(zs\) can be used, but \(xs\) and \(ys\) have been invalidated.
How does this work?

The type of the function `concat` is:

```plaintext
[a] (consumes list a, consumes list a) -> list a
```

so a call is in principle type-checked as follows:

```plaintext
(* xs @ list t * ys @ list t * ... must exist here *)
let zs = concat(xs, ys) in
(* zs @ list t * ... exist here *)
```

The available permissions `vary` with time.
How does this work?

The system knows that

- `xs @ list int` is a *duplicable* permission, whereas
- `xs @ list (ref int)` is not: it is an *affine* permission.

A caller of `concat` can give up one copy of `xs @ list int` and keep one copy. The permission is effectively *not consumed*. No such trick is possible with `xs @ list (ref int)`.

Thus, `concat` is type-checked once, but behaves differently at different call sites.
How about mutable lists?

Mutable lists support in-place meld-ing:

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

The permission \(xs @ \text{mlist t}\) is \textit{never} duplicable, regardless of the type \(t\) of the list elements, so a call to \(\text{meld(xs, ys)}\) \textit{always} invalidates the arguments \(xs\) and \(ys\).
Beyond what has been illustrated here, Mezzo has:

- permissions for composite data structures, which can be decomposed and recombined;
- permissions that express must-alias and must-not-alias information;
- a mechanism by which the existence of a permission can be ascertained at runtime.
What for?

How?

A tiny taste

Project status
The project started about one year ago and currently involves

- *Jonathan Protzenko* (Ph.D student),
- *Thibaut Balabonski* (post-doc researcher),
- and myself (INRIA researcher).
Where we are

We currently have:

• a formal definition and *soundness proof* for Core Mezzo;
• a prototype *type-checker*. 
In the short term, we would like to:

• stabilize and extend the definition of the language;
• work on type inference, which is tricky;
• write code! and evaluate the usability of the language;
• compile Mezzo down to untyped OCaml;
• work on shared-memory concurrency.
Many as-yet-unanswered questions!

- What support for *modularity*?
- What about specifications & *proofs* of programs?
- What if we lack the *manpower* to grow a new language?
- Can we *transfer* these ideas to a mainstream language?
Please find more information online at
http://gallium.inria.fr/~protzenk/mezzo-lang/
The algebraic data type of immutable lists.

```haskell
data list a =
  | Nil
  | Cons { head: a; tail: list a }
```
val rec concat [a] (consumes xs: list a, consumes ys: list a) : list a =

match xs with
| Nil  -> ys
| Cons ->
    Cons { head = xs.head;
           tail = concat (xs.tail, ys) }
end
The algebraic data type of mutable lists.

```haskell
mutable data mlist a =
  | MNil
  | MCons { head: a; tail: mlist a }
```
Mutable list concatenation

```plaintext
val rec concat1 [a] (xs: MCons { head: a; tail: mlist a }, consumes ys: mlist a) : () =
match xs.tail with
| MNil -> xs.tail <- ys
| MCons -> concat1 (xs.tail, ys)
end

val concat [a] (consumes xs: mlist a, consumes ys: mlist a) : mlist a =
match xs with
| MNil -> ys
| MCons -> concat1 (xs, ys); xs
end
```