Why design a new programming language?

The Mezzo case

François Pottier  
francois.pottier@inria.fr

Jonathan Protzenko  
jonathan.protzenko@inria.fr

INRIA

FSFMA'13
Plan

1. Some background
2. Going beyond type-checking
3. The story about state
4. Designing a type system with state
5. A glimpse of Mezzo
6. Conclusion
Type-checking: a way to reason about your programs
How do we see type-checking?

- a way of assigning types to objects, thus
- gaining static information about the memory shape of objects, while
- enabling the programmer to reason about their programs.
These properties are **static**.

You can deduce them by analyzing your program *before* running it.
How do we sell type-checking?

- The ability for the programmer to avoid bugs.
- The ability for the compiler to emit better code.
- Guarantees about safety (e.g. the program won't crash): C#, ML, Java...
Type-checking occupies a sweet spot in our landscape.
Why do we love typing so much?

- Requires no user input; the system can automatically deduce properties.
- Good properties: decidable, reasonable computational complexity.
Plan

1. Some background
2. Going beyond type-checking
3. The story about state
4. Designing a type system with state
5. A glimpse of Mezzo
6. Conclusion
How does one push a type system further?

- extend type-check more programs;
- refine provide stronger guarantees about programs.

Here are some directions that have been explored already.
Going beyond type-checking

Direction #1: the proof assistant

One may want to...

- extend the **theoretical power** of the type system;
- and **lose automation**;
- the user has to **painfully** write types by hand;
- these types are actually **proofs**.

Example:

![Coq](image-url)
One may want to...

- keep a **simple type system**;
- have a language of **pre- and post-conditions** on the side;
- delegate the task of proving to **SMT-solvers**;
- only semi-automated; SMT-solvers are **unpredictable** and not very robust.

**Example:**

![Why3](https://example.com/why3)

*Jonathan Protzenko (INRIA)*
Direction #3: Abstract interpretation

One may want to...

- design a framework to analyze the range of possible values;
- either in compilers (flags) or external tools (static analyzers).

Example: the Astrée static analyzer.
There's a whole range of possible directions.

There are some design choices that we do not wish to reproduce.

What's our « business model »? Refine the type system of ML to talk about state.
Plan

1. Some background
2. Going beyond type-checking
3. The story about state
4. Designing a type system with state
5. A glimpse of Mezzo
6. Conclusion
Most programs carry an inherent notion of state.

A socket may move from «valid socket» to «invalid socket».

Yet, no mainstream type system offers facilities for reasoning about state.
Reasoning about state…

```
let x = create_socket () in
(* x @ socket (valid) *)
let y = x in
(* x @ socket (valid), y @ socket (valid) *)
...
(* x @ socket (valid), y @ socket (valid) *)
destroy_socket x;
(* x @ socket (invalid), y @ socket (valid) *)
destroy_socket y;
(* apocalypse! *)
```
Reasoning about state is hard

Are \(x\) and \(y\) the same thing?

This is the **aliasing** problem, which is **not decidable** in general.
Question

Can we design a better type system that would:

- help the programmer reason about state, thus
- ruling out incorrect behaviors, while
- enabling new programming idioms?

This is the Mezzo project.
Plan

1. Some background
2. Going beyond type-checking
3. The story about state
4. Designing a type system with state
5. A glimpse of Mezzo
6. Conclusion
Permissions

A variable does not have a fixed type.

Instead, we may possess a permission \( x \ @ \ t \), allowing us to use \( x \) in certain ways, depending on \( t \).

This permission may disappear, to be replaced by a different one.
Immutable vs. mutable

The system maintains the following invariant:

- if $x$ is a mutable object, there exists at most one permission to read and write $x$
- if $x$ is an immutable object, there exists arbitrarily many permissions to read $x$
Why the distinction?

This distinction is central in the design of Mezzo.
- State changes become type changes.
- Since mutable objects have a unique owner, it is now safe for the type of an object to change.

This enables us to track the state of objects.
Why the distinction?

• In a **concurrent** context, the unique-owner property statically guarantees that the program is **data-race free**.

• In terms of reasoning, I can now state that **no other part** of the program may access my mutable memory. This is a **separation property**.
Plan

1. Some background
2. Going beyond type-checking
3. The story about state
4. Designing a type system with state
5. A glimpse of Mezzo
6. Conclusion
A simple example: concatenation of immutable lists.
What happens when one concatenates two immutable lists \(xs\) and \(ys\)?
This creates sharing.
Harmless sharing

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

This is harmless. We would like to accept this code.
Potentially harmful sharing

What if the lists have *mutable* elements?

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

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

We would like to *accept* this code yet *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:

\[
\begin{align*}
xs & \rightarrow \text{list int} \\
yz & \rightarrow \text{list int} \\
zs & \rightarrow \text{list int}
\end{align*}
\]

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:

```c
xs @ list (ref int)
ys @ list (ref int)
```

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

```c
zs @ 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 `append` is:

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

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

\[
(* \ xs \ @ \text{list } t \ * \ ys \ @ \text{list } t \ * \ ... \ must \ exist \ here *) \\
\text{let } zs = \text{append}(xs, \ ys) \text{ in} \\
(* \ zs \ @ \text{list } t \ * \ ... \ exist \ here *)
\]

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 `append` 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, `append` is type-checked once, but behaves differently at different call sites.
A glimpse of Mezzo

Still...how do we type-check this?

```
let x = create_socket () in
(* ? *)
let y = x in
(* ? *)
...
(* ? *)
destroy_socket x;
(* ? *)
destroy_socket y;
(* ? *)
```
A glimpse of Mezzo

Still...how do we type-check this?

```mezzo
let x = create_socket () in
(* ? *)
let y = x (* ? *)
(* ? *)
(* ? *)
... destroy_socket x;
(* ? *)
(* ? *)
destroy_socket y;
(* ? *)
```

- Keep track of local aliasing relationships.
- Declare types `valid_socket` and `invalid_socket`
- Declare `destroy_socket: (consumes x: valid_socket) -> (| x @ invalid_socket)`
A glimpse of **Mezzo**

Still...how do we type-check this?

```ocaml
let x = create_socket () in
(* x @ valid_socket *)

let y = x in
(* x @ valid_socket * x @ =y *)
...
(* x @ valid_socket * x @ =y *)

destroy_socket x;
(* x @ invalid_socket * x @ =y *)

destroy_socket y;
(* Error: could not find permission y @ valid_socket; 
   the only permissions available for it are: 
   y @ invalid_socket *)
```
The mechanisms presented so far remain relatively rigid. We offer a mechanism, called adoption/abandon, that:

- allows one to gain the freedom to alias objects, at the expense of
- paying runtime checks whenever they want to use the object.

The runtime checks guarantee that only one person owns the object. If the programmer makes a mistake, the program aborts.
An escape mechanism (2)

All type systems are a *tradeoff* between complexity and dynamic checks (Java, C++, C#...).

We drew a line: **non-tree-shaped ownership patterns** cannot be treated statically.
Plan

1. Some background
2. Going beyond type-checking
3. The story about state
4. Designing a type system with state
5. A glimpse of Mez\(\text{Zo}\)

Conclusion
The *Mezzo* language

*Mezzo* is a language that:

- takes the usual ingredients of a type system, but
- provides stronger guarantees, while still
- retaining some key properties: automated reasoning, predictability...

This is achieved through a careful blending of runtime tests / static guarantees.
The Mezzo language

 Programs written in Mezzo enjoy strong guarantees:

- the type system rules out representation exposure;
- avoids unwanted sharing;
- guarantees data-race freedom.
The *Mezzo* language

We also believe that:

- writing a program in *Mezzo* force the programmer to have a **clear understanding** of ownership,
- thus giving **better guarantees** about the program, as well as
- making it more **amenable to program proof** (long-term goal).
Conclusion

The state of Mezzo

The type system has been proved sound using the Coq proof assistant.

We have a prototype type-checker that successfully type-checks our library as well as numerous examples (several thousand lines).
Concurrency.

There are several concurrency patterns.

- How can we **axiomatize** them? (What is their type?)
- Is it sound? (Can we add these to our proof?)
- Shall we add new concurrency patterns in *Mezzo*?
Future direction #2

Inference.

Inference is a challenge; we want to limit manual intervention from the programmer, but:

- some situations require type annotations;
- can we predict which situations will require manual hints?
- can we improve our prototype with a better type-checking algorithm?
Arithmetic.

Like in ML, there are bounds-check on array accesses.

- Can we extend the permission mechanism to also talk about arithmetic?
- Can we have the type-checker perform arithmetic reasoning? (SMT-Solver)
- How viable is this approach, can we extend it beyond arithmetic?
More information

You can visit the Mezzo website

The implementation of append

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

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
  if xs then
    Cons { head = xs.head; tail = append (xs.tail, ys) }
  else
    ys
```
The (other) implementation of `append`

```plaintext
data mutable cell a =  
| Cell { head: a; tail: () } 
```

Jonathan Protzenko (INRIA)
The (other) implementation of append

```plaintext
def val rec appendAux [a] (consumes dst: cell a, consumes xs: list a, consumes ys: list a) : (| dst @ list a) =
  if xs then begin
    let dst' = Cell { head = xs.head; tail = () } in
    freeze (dst, dst');
    appendAux (dst', xs.tail, ys)
  end
  else freeze (dst, ys)
```

Jonathan Protzenko (INRIA)
val append [a] (  
    consumes xs: list a,  
    consumes ys: list a  
) : list a =  
  if xs then begin  
    let dst = Cell { head = xs.head; tail = () } in  
    appendAux (dst, xs.tail, ys);  
    dst  
  end  
else  
  ys
The implementation of **append** (mutable)

data mutable mlist a =
  | MNil
  | MCons { head: a; tail: mlist a }
The implementation of append (mutable)

```ocaml
val rec append1 [a]
  (xs: MCons { head: a; tail: mlist a },
   consumes ys: mlist a) : () =
match xs.tail with
| MNil -> xs.tail <- ys
| MCons -> append1 (xs.tail, ys)
end

val append [a] (consumes xs: mlist a,
    consumes ys: mlist a) : mlist a =
match xs with
| MNil -> ys
| MCons -> append1 (xs, ys); xs
end
```