Mezzo

a typed language for safe and effectful concurrent programs

Jonathan Protzenko (INRIA)
jonathan.protzenko@inria.fr
This defense:

1. some context;
2. the design of *Mezzo*;
3. the implementation of *Mezzo*.
Some context
In fact, my main conclusion after spending ten years of my life working on the TeX project is that software is hard.

Don. Knuth
In fact, my main conclusion after spending ten years of my life working on the \TeX{} project is that software is hard.

Don. Knuth, author of \TeX
How can we make writing software easier?

A natural idea is to use the computer to verify the absence of certain errors.
How can we make writing software easier?

A natural idea is to use the computer to verify the absence of certain errors.

```ocaml
# let years_in_phd = 4 in
  if years_in_phd = "too long" then
    print_endline "oops";;
```

Error: The function `=', expects 2 arguments of types ['a] and ['a], but it is given 2 arguments of types [int] and [string].
How can we make writing software easier?

A natural idea is to use the computer to verify the absence of certain errors.

```plaintext
# let years_in_phd = 4 in
if years_in_phd = "too long" then
    print_endline "oops";;
```

Error: The function `=' expects 2 arguments of types ['a] and ['a], but it is given 2 arguments of types [int] and [string].

The error is identified in advance: the compiler rejects the program.
Have you met... type systems?

A type system assigns types to expressions; it makes sure we don’t mix `int` and `string`.

The point is to ensure memory safety. Indeed, well-typed programs do not exhibit memory errors.
Type systems are imperfect

The type system can’t check everything.

```ocaml
# let oc = open_out "/tmp/journal";;
# close_out oc;;
# output_string oc "Dear journal...";;
Exception: Sys_error "Bad file descriptor".
```

The error arises too late: the compiler has accepted the program, yet the program executes, and runs into an error.
Type systems are imperfect

The type system can’t check everything.

```ocaml
# let oc = open_out "'/tmp/journal";;
# close_out oc;;
# output_string oc "Dear journal...";;
Exception: Sys_error "Bad file descriptor".
```

The error arises too late: the compiler has accepted the program, yet the program executes, and runs into an error.

There is a rich design space to explore.
It’s all about the balance!
With great power, comes great complexity.
Let’s explore the issue.
What kind of type language?

```ocaml
let r = ref 0
let uniq =
  fun () ->
    r := !r + 1;
    !r
```

A weak type for `uniq` is (ML):

```
unit \rightarrow \text{int}
```

A strong type for `uniq` is (proof):

- requires:  \( r : \text{ref int} \)
- ensures: \( r : \text{ref int} \land \) 
  \( \text{old}(r.\text{contents}) + 1 = r.\text{contents} \land \)
  \( \text{ret} = r.\text{contents} \)
Mezzo is a language with a stronger type system that tries to talk about ownership, hence providing better support for modular reasoning.
What is ownership?

A way to classify what I and others can do with a piece of data.
The kind of issues we want to tackle

- Will this function modify this global, shared reference?
- Can I make sure two threads don’t race for the same memory cell?
- Is this list still usable after a function call?
- Is it safe to let the client manipulate my internal list of items?

These questions all revolve around the concept of ownership.

Ownership is crucial, but the type system of ML does not talk about it.
The *Mezzo* style of typing

```
let r = ref 0
let uniq =
  fun () ->
   r := !r + 1;
   !r
```

The *Mezzo* type system says `uniq` has type:

```
(| r@ref int) → int
```

Definitely not your run-of-the-mill ML type system, but not quite program proof either.
The **Mezzo** style of typing (2)

```ml
let r = ref 0
let uniq =
  fun () ->
    r := !r + 1;
    !r
let x₁, x₂ = uniq() || uniq()
```

ML says “ok”. But there’s a **race condition**, and **Mezzo** rejects this program.

**In fact, **Mezzo** programs are **data-race** free!
The present thesis
Main contributions

• A carefully-designed language

• Novel type-theoretic mechanisms

• A matching implementation
Let’s jump in!
Mezzo is not ML

Mezzo has permissions, of the form $x \, @ \, t$, separated by $\ast$.

In ML:
\[
\Gamma = x : t, y : u
\]
In Mezzo:
\[
P = x \, @ \, t \ast y \, @ \, u
\]

```ml
val f (x: ...): ... =
  let y = ... in
  ...
```
Mezzo is not ML

Mezzo has permissions, of the form $x \circ t$, separated by $\ast$.

In ML: $\Gamma = x : t, y : u$

In Mezzo: $P = x \circ t \ast y \circ u$

$$\text{val f (x: ...): ... =}$$
$$\text{let y = ... in}$$
$$\text{...}$$
Mezzo is not ML

Mezzo has permissions, of the form \( x @ t \), separated by \(*\).

In ML: \( \Gamma = x : t, y : u \)

In Mezzo: \( P = x @ t \ast y @ u \)

\[
\begin{align*}
\text{val } f ( x : \ldots ) : \ldots &= \\
\text{let } y = \ldots \text{ in } \\
\ldots
\end{align*}
\]
Mezzo is not ML

Mezzo has permissions, of the form $x @ t$, separated by $\ast$.

\[
\begin{align*}
\text{In ML:} & \quad \Gamma = x : t, y : u \\
\text{In Mezzo:} & \quad P = x @ t \ast y @ u
\end{align*}
\]

\[
\text{val f (x: ...): ... =}
\]

\[
\begin{align*}
\text{let y = ... in} \\
\text{...}
\end{align*}
\]

\[
P_3
\]
Different *modes* for types

<table>
<thead>
<tr>
<th></th>
<th>duplicable</th>
<th>exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l )</td>
<td>read-only</td>
<td>read-write</td>
</tr>
<tr>
<td>others</td>
<td>read-only</td>
<td>—</td>
</tr>
</tbody>
</table>

Depends on the definition of \( t \):
- `list int` is *duplicable* because *immutable*
- `ref int` is *exclusive* because *mutable*

This is a *design choice*. The *user story* is simple: mutable = unique, immutable = shared.

Asserts *ownership* of a fraction of the heap.
Function may **consume** ownership of their arguments.

```merlin
val append : [a] (  
  consumes xs : list a,  
  consumes ys : list a  
) -> (zs : list a)
```

Mezzo: a language with permissions
Function may **consume** ownership of their arguments.

```plaintext
val append: [a] (  
  consumes xs: list a,
  consumes ys: list a 
) -> (zs: list a)
```

Let’s see explain concatenation *visually.*
Concatenation may be dangerous because it creates sharing. What about:

\[
\text{iter\_incr } xs \parallel \text{ iter\_incr } zs
\]

How can we make this safe?
Back to the signature.

```
val append: [a] (  
    consumes xs: list a,  
    consumes ys: list a  
) -> (zs: list a)
```
Example: list (ref int).

\[ \text{let } zs = \text{append } (xs, ys) \text{ in } \]

\[ \ldots \]
**Mezzo: a language with permissions**

Example:

```ml
let zs = append (xs, ys) in
```

Before function call:

- `xs @ list (ref int) *`  
- `ys @ list (ref int)`
Example: \texttt{list (ref int)}.

\begin{verbatim}
... 
let zs = append (xs, ys) in
...
\end{verbatim}

After function call
\begin{verbatim}
zs @ list (ref int)
\end{verbatim}
Mezzo: a language with permissions

Example: list int.

... let zs = append (xs, ys) in ...
...
Mezzo: a language with permissions

Example:

```
let zs = append (xs, ys) in
```

Before function call

```
xs @ list int * ys @ list int
```

...
Example: list int.

... let zs = append (xs, ys) in ...

After function call
xs @ list int * ys @ list int *
zs @ list int
Complete example: type-checking `append`
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
  match xs with
  | Cons { head = h; tail = t } ->
    let t' = append (t, ys) in
    Cons { head = h; tail = t' }
  | Nil ->
    ys
  end
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =

match xs with
| Cons { head = h; tail = t } ->
  let t' = append (t, ys) in
  Cons { head = h; tail = t' }
| Nil ->
  ys
end

Permissions
ys @ list a *
xs @ list a
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
  match xs with
  | Cons { head = h; tail = t } ->
    let t' = append (t, ys) in
    Cons { head = h; tail = t' }
  | Nil ->
    ys
  end

Permissions

ys @ list a *
xs @ Cons { head = h; tail = t } *
h @ a * t @ list a
open list

val rec append [a] (xs @ Cons { head = h; tail = t } *, ys @ list a *) : list a =
  match xs with
  | Cons { head = h; tail = t } ->
    let t' = append (t, ys) in
    Cons { head = h; tail = t' }
  | Nil ->
    ys
  end

Permissions

ys @ list a *
xs @ Cons { head = h; tail = t } *
h @ a * t @ list a
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
  match xs with
  | Cons { head = h; tail = t } ->
    let t' = append (t, ys) in
    Cons { head = h; tail = t' }
  | Nil ->
    ys
  end
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
  match xs with
  | Cons { head = h; tail = t } ->
    let t' = append (t, ys) in
    Cons { head = h; tail = t' }
  | Nil ->
    ys
  end

Permissions

ys @ list a *
xs @ Cons { head = h; tail = t } *
h @ a * t @ list a
open list

val rec append [a] (consumes xs: list a,
consumes ys: list a)
): list a =
match xs with
| Cons { head = h; tail = t } ->
   let t' = append (t, ys) in
   Cons { head = h; tail = t' }
| Nil ->
   ys
end
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
match xs with
| Cons { head = h; tail = t } ->
  let t' = append (t, ys) in Cons { head = h; tail = t' }
| Nil ->
  ys
end

Permissions

ys @ list a *
xs @ Cons { head = h; tail = t } *
h @ a * t @ list a
t' @ list a
ret @ Cons {
  head = h;
  tail = t'
}
open list

val rec append [a] (  
consumes xs: list a,  
consumes ys: list a
): list a =  
match xs with  
| Cons { head = h; tail = t } ->  
  let t' = append (t, ys) in  
  Cons { head = h; tail = t' }  
| Nil ->  
  ys
end
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
  match xs with
  | Cons { head = h; tail = t } ->
    let t' = append (t, ys) in
    Cons { head = h; tail = t' }
  | Nil ->
    ys
  end

Permissions

ys @ list a *
xs @ Cons { head = h; tail = t } *
h @ a * t @ list a
t' @ list a
ret @ Cons {
  head = h;
  tail: list a
}
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
  match xs with
  | Cons { head = h; tail = t } ->
    let t' = append (t, ys) in
    Cons { head = h; tail = t' }
  | Nil ->
    ys
end

Permissions

ys @ list a *
xs @ Cons { head = h; tail = t } *
h @ a *
t @ list a
t' @ list a
ret @ Cons {
  head = h;
  tail: list a}
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
match xs with
| Cons { head = h; tail = t } ->
  let t' = append (t, ys) in
  Cons { head = h; tail = t' }
| Nil ->
  ys
end
open list

val rec append [a] (consumes xs: list a,
consumes ys: list a): list a =
  match xs with
  | Cons { head = h; tail = t } ->
    let t' = append (t, ys) in
    Cons { head = h; tail = t' }
  | Nil ->
    ys
end

Permissions

ys @ list a *
xs @ Cons { head = h; tail = t } *
h @ a * t @ list a
t' @ list a
ret @ Cons {
  head: a;
  tail: list a
}
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
  match xs with
  | Cons { head = h; tail = t } ->
    let t' = append (t, ys) in
    Cons { head = h; tail = t' }
  | Nil ->
    ys
  end

Permissions

ys @ list a *
xs @ Cons { head = h; tail = t } *
h @ a * t @ list a
t' @ list a
ret @ Cons {
  head: a;
  tail: list a
}
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
  match xs with
  | Cons { head = h; tail = t } ->
    let t' = append (t, ys) in
    Cons { head = h; tail = t' }
  | Nil ->
    ys
  end

Permissions

ys @ list a *
xs @ Cons { head = h; tail = t } *
h @ a *
t @ list a
t' @ list a
ret @ list a
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
  match xs with
  | Cons { head = h; tail = t } ->
    let t' = append (t, ys) in
    Cons { head = h; tail = t' }
  | Nil ->
    ys
end
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
  match xs with
  | Cons { head = h; tail = t } ->
    let t' = append (t, ys) in
    Cons { head = h; tail = t' }
  | Nil ->
    ys
end

Permissions
ys @ list a *
xs @ Nil *
ret = ys
open list

val rec append [a] (  
  consumes xs: list a,  
  consumes ys: list a  
): list a =  
  match xs with  
  | Cons { head = h; tail = t } ->  
    let t' = append (t, ys) in  
    Cons { head = h; tail = t' }  
  | Nil ->  
    ys  
end  

Permissions  
ys @ list a *  
xs @ Nil *  
ret = ys
open list

val rec append [a] (consumes xs: list a, consumes ys: list a): list a =
  match xs with
  | Cons { head = h; tail = t } ->
    let t' = append (t, ys) in
    Cons { head = h; tail = t' }
  | Nil ->
    ys
  end
The base layer
Mezzo is definitely not ML

Singleton types \( x @ (=y): x \text{ is } y \)
Written as: \( x = y \)

Constructor types \( xs @ \text{Cons} \{ \text{head}: t; \text{tail}: u \} \)
(special-case: \( t \) is a singleton, we write \( xs @ \text{Cons} \{ \text{head} = \ldots; \text{tail} = \ldots \} \))

Decomposition via unfolding (named fields), refinement (matching) and folding (subtyping)

Several possible types \( x @ (\text{int, int}), \)
\( x @ \exists(y,z: \text{value}). \)
\( (=y | y @ \text{int}, =z | z @ \text{int}) \),
\( x @ \exists t.t, \text{etc.} \)
A glance at the type-checking rules

General form: $K, P \vdash e : t$. ($K =$ kinding environment)

Sub

\[ K; P_2 \vdash e : t_1 \]
\[ P_1 \leq P_2 \quad t_1 \leq t_2 \]
\[ K; P_1 \vdash e : t_2 \]

Frame

\[ K; P_1 \vdash e : t \]
\[ K; P_1 \cdot P_2 \vdash e : (t \mid P_2) \]

Read

$t$ is duplicable

\[ P \text{ is } x @ A \{ \ldots ; f : t ; \ldots \} \]
\[ K; P \vdash x.f : (t \mid P) \]

Tuple

\[ K \vdash (x_1, \ldots , x_n) : (=x_1, \ldots , =x_n) \]

Application

\[ K; x_1 @ t_2 \rightarrow t_1 \ast x_2 @ t_2 \vdash x_1 \ x_2 : t_1 \]
A glance at the subsumption relation

\[
\text{DecomposeTuple} \\
\quad y @ (. . . , t, . . . ) \\
\equiv \exists ( x : \text{value}) ( y @ (. . . , =x, . . . ) \ast x @ t )
\]

\[
\text{EqualsForEquals} \\
( y_1 = y_2 ) \ast [ y_1 / x ] P \equiv ( y_1 = y_2 ) \ast [ y_2 / x ] P
\]

\[
\text{EqualityReflexive} \\
\text{empty} \leq ( x = x )
\]

\[
\text{Fold} \\
A \{ \bar{f} : \bar{t} \} \text{ is an unfolding of } X \bar{\bar{T}} \\
x @ A \{ \bar{f} : \bar{t} \} \leq x @ X \bar{\bar{T}}
\]
Explaining the design choices

Singleton types allow us to keep track of equalities within the type system: unified, regular approach

Concrete types a.k.a. “constructor” types implement refinement and state change: new patterns

Subsumption is the key ingredient that allows to use any representation interchangeably
The dynamic layer
An example that breaks

We need to represent a graph.

Imagine a DFS. We need to mark (mutable) nodes.

```haskell
data node = mutable Node {
    neighbors: list node;
    seen:  bool;
}

data graph = mutable Graph {
    roots: list node;
}

val g: graph =
    let n = Node { neighbors = nil; seen = false } in
    n.neighbors <- cons (n, nil);
    Graph { neighbors = cons (n, nil) }
```
data node = mutable Node {
    neighbors: list node;
    seen: bool;
}

data graph = mutable Graph {
    roots: list node;
}

val g: graph =
  let n = Node { neighbors = nil; seen = false } in
  let neighbors = cons (n, nil) in
  n.neighbors <- neighbors;
  Graph { neighbors = cons (n, nil) }
data node = mutable Node { 
    neighbors: list node;
    seen: bool;
}

data graph = mutable Graph { 
    roots: list node;
}

val g: graph = 
    let n = Node { neighbors = nil; seen = false } in 
    let neighbors = cons (n, nil) in 
    n.neighbors <- neighbors; 
    Graph { neighbors = cons (n, nil) }
data node = mutable Node { 
    neighbors: list node;
    seen: bool;
}

data graph = mutable Graph { 
    roots: list node;
}

val g: graph = 
    let n = Node { neighbors = nil; seen = false } in 
    let neighbors = cons (n, nil) in 
    n.neighbors <- neighbors; 
    Graph { neighbors = cons (n, nil) }
data node = mutable Node {
    neighbors: list node;
    seen: bool;
}

data graph = mutable Graph {
    roots: list node;
}

val g: graph =
    let n = Node { neighbors = nil; seen = false } in
    let neighbors = cons (n, nil) in
    n.neighbors <- neighbors;
    Graph { neighbors = cons (n, nil) }
data node = mutable Node { 
    neighbors: list node;
    seen: bool;
}

data graph = 
    roots: list node;
}

val g: graph = 
    let n = Node { neighbors = nil; seen = false } in 
    let neighbors = cons (n, nil) in 
    n.neighbors <- neighbors; 
    Graph { neighbors = cons (n, nil) }
data node = mutable Node {  
  neighbors: list node;  
  seen: bool;  
}

data graph = mutable Graph {  
  roots: list node;  
}

val g: graph =  
  let n = Node { neighbors = nil, seen = false } in  
  let neighbors = cons (n, nil) in  
  n.neighbors <- neighbors;  
  Graph { neighbors = cons (n, nil) }
Uniqueness guaranteed via a runtime test
Uniqueness guaranteed via a runtime test
Uniqueness guaranteed via a runtime test.
Uniqueness guaranteed via a runtime test.
Uniqueness guaranteed via a runtime test
Uniqueness guaranteed via a runtime test.
Uniqueness guaranteed via a runtime test
Uniqueness guaranteed via a runtime test.
Uniqueness guaranteed via a runtime test
Uniqueness guaranteed via a runtime test.
Uniqueness guaranteed via a runtime test
The dynamic solution

data mutable node =
    Node {
        contents : int;
        visited  : bool;
        neighbors: list dynamic;
    }

data mutable graph =
    Graph {
        roots   : list dynamic;
    } adopts node
The dynamic solution

```haskell
data mutable node =
  Node {
    contents : int;
    visited : bool;
    neighbors: list dynamic;
  }

data mutable graph =
  Graph {
    roots : list dynamic;
  } adopts node
```

The dynamic type

List of pointers without ownership
The dynamic solution

```
data mutable node = Node {
    contents : int;
    visited : bool;
    neighbors: list dynamic;
}
```

```
data mutable graph = Graph {
    roots : list dynamic;
} adopts node
```

Adoption: The graph object owns the nodes.
val g : graph = let n = Node {
  contents = 10;
  visited = false;
  neighbors = ();
} in
let ns = cons [dynamic] (n, nil) in
n.neighbors <- ns;
let g = Graph { roots = ns } in
give n to g;
g
val g : graph =
let n = Node {
    contents = 10;
    visited = false;
    neighbors = ();
} in
let ns =
    cons [dynamic] (n, nil)
in
n.neighbors <- ns;
let g = Graph { roots = ns } in
give n to g;
g
val g : graph = 
let n = Node { 
  contents = 10; 
  visited = false; 
  neighbors = (); 
} in 
let ns = 
  cons [dynamic] (n, nil) 
in 
n.neighbors <- ns; 
let g = Graph { roots = ns } in 
give n to g;
g
val g : graph = 
  let n = Node {
    contents = 10;
    visited = false;
    neighbors = ();
  } in
  let ns =
    cons [dynamic] (n, nil)
in
  n.neighbors <- ns;
  let g = Graph { roots = ns } in
  give n to g;
  g

Permissions
n @ Node {
  contents: int; visited: bool;
  neighbors: ()
} *
ns @ list dynamic
val g : graph = 
let n = Node {
  contents = 10;
  visited = false;
  neighbors = ();
} in
let ns =
  cons [dynamic] (n, nil)
in
n.neighbors <- ns;
let g = Graph { roots = ns } in
give n to g;
g
val g : graph = 
let n = Node {
    contents = 10;
    visited = false;
    neighbors = ();
} in
let ns = 
    cons [dynamic] (n, nil)
in
n.neighbors <- ns;
let g = Graph { roots = ns } in
give n to g;
g

Permissions
n @ Node {
    contents: int; visited: bool;
    neighbors = ns
} *
n @ dynamic *
ns @ list dynamic *
g @ Graph { roots = ns }
val g : graph = 
  let n = Node {
    contents = 10;
    visited = false;
    neighbors = ();
  } in
let ns =
  cons [dynamic] (n, nil)
in
n.neighbors <- ns;
let g = Graph { roots = ns } in
give n to g;
g

Permissions
n @ Node {
  contents: int; visited: bool;
  neighbors = ns
} *
  n @ dynamic *
  ns @ list dynamic *
g @ Graph { roots = ns }
val g : graph =
  let n = Node {
    contents = 10;
    visited = false;
    neighbors = ();
  } in
  let ns =
    cons [dynamic] (n, nil)
  in
  n.neighbors <- ns;
  let g = Graph { roots = ns } in
  give n to g;
  g

Permissions
n @ Node {
  contents: int; visited: bool;
  neighbors = ns
}

n @ dynamic *

ns @ list dynamic *

g @ Graph { roots = ns }
val g : graph = 
  let n = Node { 
    contents = 10; 
    visited = false; 
    neighbors = (); 
  } in 
  let ns = cons [dynamic] (n, nil) in 
  n.neighbors <- ns; 
  let g = Graph { roots = ns } in 
  give n to g; 
  g

Permissions
n @ Node { 
  contents: int; visited: bool; 
  neighbors = ns 
} *
ns @ list dynamic *
g @ Graph { roots: list dynamic }
val g : graph = 
let n = Node {
    contents  = 10;
    visited   = false;
    neighbors = ();
} in
let ns = 
    cons [dynamic] (n, nil)
in
n.neighbors <- ns;
let g = Graph { roots = ns } in
give n to g;
g
val g : graph = 
let n = Node {
    contents = 10;
    visited = false;
    neighbors = ()
} in
let ns =
    cons [dynamic] (n, nil)
in
n.neighbors <- ns;
let g = Graph { roots = ns } in
give n to g;
g

 Permissions

n @ Node {
    contents: int; visited: bool;
    neighbors = ns
} *

n @ dynamic *

ns @ list dynamic *
g @ graph
val g : graph = let n = Node {
  contents = 10;
  visited = false;
  neighbors = ();
} in let ns = cons [dynamic] (n, nil) in n.neighbors <- ns; let g = Graph { roots = ns } in give n to g; g

Permissions
n @ Node {
  contents: int; visited: bool;
  neighbors: list dynamic
}
ns @ list dynamic *
g @ graph
val g : graph = 
let n = Node {
  contents = 10;
  visited = false;
  neighbors = ();
} in 
let ns =
  cons [dynamic] (n, nil)
in
n.neighbors <- ns;
let g = Graph { roots = ns } in
give n to g;
g
val g : graph = 
  let n = Node {
    contents = 10;
    visited = false;
    neighbors = ();
  } in
  let ns =
    cons [dynamic] (n, nil)
  in
  n.neighbors <- ns;
  let g = Graph { roots = ns } in
  give n to g;
  g
val g : graph =
  let n = Node {
    contents = 10;
    visited = false;
    neighbors = ()
  } in
let ns =
  cons [dynamic] (n, nil)
in
n.neighbors <- ns;
let g = Graph { roots = ns } in
give n to g;
g

Permissions
n @ node *
n @ dynamic *
ns @ list dynamic *
g @ graph
val g : graph = 
  let n = Node {
    contents = 10;
    visited = false;
    neighbors = ();
  } in
  let ns =
    cons [dynamic] (n, nil)
  in
  n.neighbors <- ns;
  let g = Graph { roots = ns } in
  give n to g;
  g
A glance at the typing rules

\[ x = \text{adoptee}, \ y = \text{adopter} \]

**Give**

\[
\frac{t_2 \ \text{adopts} \ t_1}{K; x @ t_1 \ast y @ t_2 \vdash \text{give } x \ \text{to} \ y : (\mid y @ t_2)}
\]

**Take**

\[
\frac{t_2 \ \text{adopts} \ t_1}{K; x @ \text{dynamic} \ast y @ t_2 \vdash \text{take } x \ \text{from} \ y : (\mid x @ t_1 \ast y @ t_2)}
\]
Reflecting on the design of adoption/abandon

<table>
<thead>
<tr>
<th></th>
<th>run-time check</th>
<th>two-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>static regions</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>nesting</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>locks</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>adoption/abandon</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Adoption/abandon is another **essential** contribution of *Mezzo*.
Looking back on adoption/abandon

This mechanism bridges the static and dynamic disciplines.

It allows one to take two elements out at the same time.

It provides a built-in, efficient mechanism for fulfilling the proof obligation $x_1 \neq x_2$ using a run-time test.
The implementation of adoption/abandon

Each object in the heap has a hidden field. Each adoptee maintains the address of its adopter in the hidden field.

- **give x to y** writes the address of y in the hidden field of x
- **take x from y** compares the address of y with the hidden field of x; if match, writes NULL in the hidden field of x
Looking back on adoption/abandon (2)

This may seem simple; the final version is the product of many iterations and many attempts.

One advantage: the name of the *adopter* serves as the name of the *conceptual* region for the adoptees. (Usability!)

The proof of soundness guarantees that adoption/abandon is safe (F. Pottier).
Type-checking MezzO
A glance at the subsumption relation (2)

ForallElim
\[ \forall (X : \kappa) \; P \leq [T/X]P \]

CopyDups
\[ P \text{ is duplicable} \]
\[ C[t] \ast P \leq C[ (t \mid P) ] \ast P \]

HideDuplicablePrecondition
\[ P \text{ is duplicable} \]
\[ (x @ (t_1 \mid P) \rightarrow t_2) \ast P \leq x @ t_1 \rightarrow t_2 \]

ExistsIntro
\[ [T/X]P \leq \exists (X : \kappa) \; P \]

CoArrow
\[ u_1 \leq t_1 \quad t_2 \leq u_2 \]
\[ x @ t_1 \rightarrow t_2 \leq x @ u_1 \rightarrow u_2 \]

Unfold
\[ A \{ \vec{f} : \vec{t} \} \text{ is an unfolding of } X \vec{T} \]
\[ X \vec{T} \text{ has only one branch} \]
\[ x @ X \vec{T} \leq x @ A \{ \vec{f} : \vec{t} \} \]
A suitable representation of permissions

*Mezzo* is a powerful language: the type-checker is complex, because of the interaction between:

* duplicable vs. non-duplicable permissions,
* equivalent permissions:
  \[ z \mathrel{\@} (\_\_\_\_x, \_\_\_\_y) \times x \mathrel{\@} \text{ref int} \times y \mathrel{\@} \text{ref int} \equiv z \mathrel{\@} (\text{ref int, ref int}), \]
* inference (of type application): \text{cons} [\_\_\_\_?] (x, y),
* subtyping:
  \[ [a] \text{ duplicable a => (ref a)} \rightarrow a \equiv [y: \text{value}] (\text{ref (=y)}) \rightarrow (=y), \]
* the frame rule...
A procedure for rewriting a permission into a normal form. In essence:

- permissions are **maximally expanded** (+ one-branch, functions),
- existential quantifiers are **opened as rigid variables**, 
- redundant conjunctions are **simplified**, 
- nested permissions are **flattened**.
Normalization rules can be applied in any order. They operate on the current permission, that is, the hypothesis.

Normalization rules decompose non-atomic permissions into atomic constructs. That is, they decompose positive connectives which are left-invertible.

These are standard proof search techniques.
**Type-checking vs. logic**

Mezzo remains a type system.

- far less connectives and rules
- \( f \, @ \, t \rightarrow u \times x \, @ \, t \not\supseteq \exists (y : \text{value}) \, y \, @ \, u \) (no implicitly callable ghost functions)
- no built-in disjunction (only tagged sums)

Mezzo’s type system feels like a limited fragment of intuitionistic logic.
The main type-checking algorithm

- A forward, flow-sensitive algorithm.
- Threads a normalized permission through program points.
- Relies on two algorithms: subtraction (deciding subtyping) and merge (simplifying disjunctions)
Subtraction: an unusual algorithm

- Subtyping needs to be decided for function calls and for function bodies.
- Blurs the frontier between type-checking and logics.
- The subtyping algorithm has to perform inference.
More about subtraction

The operation is written \( P - Q = R \).

This means:
“with the instantiation choices from \( \forall' \), we get \( P \leq Q \times R \).”
\( \mathcal{R} \) denotes rigidly-bound variables.

\[
\begin{align*}
\mathcal{R}(\ell, h, t).
\ell @ \text{Cons} \{ \text{head} = h; \text{tail} = t \} * \\
h @ \text{ref int} * t @ \text{list (ref int)} \\
- \\
\ell @ \text{list (ref int)} \\
= \\
\mathcal{R}(\ell, h, t).
\ell @ \text{Cons} \{ \text{head} = h; \text{tail} = t \}
\end{align*}
\]
Backtracking

Inference uses flexible ($\mathcal{F}$) variables.

There may be several solutions:

$$\mathcal{R}(x), \mathcal{F}(\alpha). (x \mathbin{@} \text{int} - x \mathbin{@} \alpha) = \begin{cases} 
\mathcal{R}(x)\mathcal{F}(\alpha = \text{int}) \\
\mathcal{R}(x)\mathcal{F}(\alpha = =x) & x \mathbin{@} \text{int} \\
\mathcal{R}(x)\mathcal{F}(\alpha = \text{unknown}) 
\end{cases}$$

Not all solutions are explored: $\alpha$ could be $(\beta \mid p)$.

Plus, there are other backtracking points (quantifiers).
The prototype

Backtracking stops at the expression level: we keep one solution when type-checking an expression.

The implementation relies on:
- efficient (good complexity) and easy-to-use (persistent) data structures for inference with backtracking (union-find, levels)
- fine-tuned heuristics (prioritize more likely solutions first)

Both required significant effort.
Other type-checking difficulties

data \( t = \text{mutable} \ T \)

\[
\begin{array}{c}
\begin{array}{c}
\text{T} \\
\text{x}
\end{array} \\
\begin{array}{c}
_0 \\
_1
\end{array}
\end{array}
\lor
\begin{array}{c}
\begin{array}{c}
\text{T} \\
\text{T}
\end{array} \\
\begin{array}{c}
_0 \\
_1
\end{array}
\end{array} = ?
\]
The merge operation

The merge problem arises when type-checking disjunctions (if-then-else, match).

- Combination of where to assign non-duplicable data, subtyping, graph reconstruction.
- Does not always admit a principal solution.
- Graph-based algorithm gives good results in practice.

The merge operation is less of a problem than inference difficulties.
Looking back on *Mezzo*
What we’ve learned

- Ownership as an **atomic**, fundamental concept.
- Power of a **unified approach**.
- Importance of the **surface language**.
- **Key ingredient**: the adoption/abandon approach.
- Role of **examples**.
Going further

- **Restrict** the expressivity of the system (results/usability).
- Re-use the “pluggable” approach idea (static or dynamic).
- **Extra mechanisms** for common programming patterns.
- Make the system **gradual** for better interoperability and conversion.
- **MezZo** as an extension of ML (refinement types?)
Mezzo
the language of the future

The end.
Online demo!

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

Buggy code:

```haskell
val r = newref 0
val print_uniq (| r @ ref int): () =
  r := !r + 1;
  print !r
val _ =
  thread::spawn print_uniq;
  thread::spawn print_uniq;
```

Result:

Here's a tentatively short, potentially misleading error message:

File "/tmp/test.mz", line 7, characters 16-26:

```
thread::spawn print_uniq;
^^^^^^^^^^^^
```

Could not obtain the following permission:
```
  r @ ref::ref int::int
```
Detecting race-conditions (2)

Fixed code:

```cpp
val r = newref 0
val l: lock::lock (r @ ref int) = lock::new ()
val print_uniq (): () =
  lock::acquire l;
  r := !r + 1;
  print !r;
  lock::release l
val _ =
  thread::spawn print_uniq;
  thread::spawn print_uniq;
```
DFS (in surface syntax)

(* Assumes all the nodes in the graph are set to [false]. *)

```ocaml
val traverse (g: graph): () =
  let rec visit (n: dynamic | g @ graph): () =
    take n from g;
    if n.seen then
      (* The node has been visited already *)
      give n to g
    else begin
      (* The node has not been visited yet. *)
      let neighbors = n.neighbors in
      (* Mark it as visited. *)
      n.seen <- true;
      (* We keep a copy of [children] (list dynamic is duplicable). *)
      give n to g;
      (* Recursively visit the children. *)
      list::iter (neighbors, visit)
    end
  in
  (* Visit each of the roots. *)
  iter (g.roots, visit)
```
Tail-recursive concatenation

```ocaml
data mutable xlist a =
    | XNil
    | XCons { head: a; tail: () }

alias xcons a =
    XCons { head: a; tail: () }

val rec appendAux [a] (consumes (dst: xcons a, xs: list a, ys: list a)) : (| dst @ list a)
    =
    match xs with
    | Cons ->
        let dst' = XCons { head = xs.head; tail = () } in
        freeze (dst, dst');
        appendAux (dst', xs.tail, ys)
    | Nil ->
        freeze (dst, ys)
end
```
Tail-recursive concatenation (2)

```ocaml
val append [a] (consumes (xs: list a, ys: list a)) : list a =
  match xs with
  | Cons ->
    let dst = XCons { head = xs.head; tail = () } in
    appendAux (dst, xs.tail, ys);
    dst
  | Nil ->
    ys
end
```