The design of *Mezzo*, a new programming language

François Pottier
francois.pottier@inria.fr

Jonathan Protzenko
jonathan.protzenko@inria.fr

INRIA

ICFP'13

An overview

The *Mezzo* project is about designing a new programming language.

Mezzo feels like ML, but blends existing ideas from the literature to build a type system that talks about state.

One can think of *Mezzo* as "separation logic turned into a type system, for ML". And more.

Why design a new programming language?

We want to reject dangerous programs (data races, unwanted sharing).

We want to accept more programs (progressive initialization, type-changing updates).

We posit that a strict type system makes programs more amenable to formal reasoning.

Our contribution

1. A careful blend of ideas makes up the type system (base layer).
2. A mechanism of runtime tests complements the static discipline (“dynamic” layer).

The type system of *Mezzo*

The core concept in *Mezzo* is that of a permission.

A permission `x @ t` represents the right to use `x` as a variable of type `t`.

(Read: « `x` is a `t` » or « `x` has type `t` ».)

Permissions

This is *almost* like ML.

In ML we use a typing context such as
$x : t,\ y : u$

In Mezzo we use a current permission such as
`x @ t * y @ u`

In other words, permissions are our type system.

The * connective denotes the *conjunction* of permissions.
Think separation logic.

Almost?

Permissions *come and go*.

```
let x = ref 0 in
x := true;
```

Almost?

Permissions *come and go*.

P

```
let x = ref 0 in
x := true;
```

Almost?

Permissions *come and go*.

```
let x = ref 0 in
x := true;
```

x @ ref int * P

Almost?

Permissions *come and go*.

```
let x = ref 0 in
x := true;
```

x @ ref bool * P

Almost?

Permissions *come and go*.

```
let x = ref 0 in
x := true;
```

We traded **`x @ ref int`** for **`x @ ref bool`**. This is the way *Mezzo* keeps track of state changes (*strong update*).

Almost?

Permissions *come and go*.

```
let x = ref 0 in
x := true;
```

We traded **`x @ ref int`** for **`x @ ref bool`**. This is the way *Mezzo* keeps track of state changes (*strong update*).

We thus need a notion of ownership; this implies keeping track of aliasing.

Ownership

- Permissions that denote mutable data are uniquely-owned, and grant read-write access. They are exclusive.
- Permissions that denote immutable data are shared, and grant read-only access. They are duplicable.
- Permissions that are neither exclusive or duplicable are affine.

> A permission `x @ t` represents the ownership of a fragment of the heap denoted by `t`.

Ownership

- Permissions that denote mutable data are uniquely-owned, and grant read-write access. They are exclusive.
- Permissions that denote immutable data are shared, and grant read-only access. They are duplicable.
- Permissions that are neither exclusive or duplicable are affine.

A permission `x @ t` represents the ownership of a fragment of the heap denoted by `t`.

Ownership reasoning is essential in a concurrent setting.

An example

Everyone knows the **map** function.

```
val map [a, b] (list a, a -> b) -> list b
```

An example

Everyone knows the **map** function.

```
val map [a, b] (list a, a -> b) -> list b

(* Classical OCaml version. *)
let map f = function
| [] -> []
| x :: xs -> f x :: map f xs
```

An example

Everyone knows the **map** function.

```
val map [a, b] (list a, a -> b) -> list b
```

```
(* Classical OCaml version. *)
let map f = function
| [] -> []
| x :: xs -> f x :: map f xs
```

The ML version is not tail-recursive.

An example

Everyone knows the **map** function.

```
val map [a, b] (list a, a -> b) -> list b
```

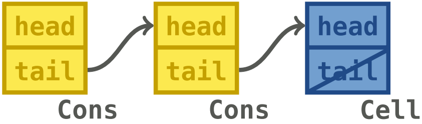
```
(* Classical OCaml version. *)
let map f = function
| [] -> []
| x :: xs -> f x :: map f xs
```

The ML version is not tail-recursive.

Let us leverage *Mezzo* to write a tail-recursive version.

Tail-recursive map

This code cannot be written in ML.

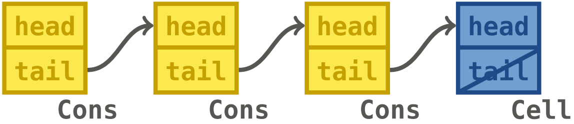


Cons blocks are immutable.
Cell blocks are mutable.

Cons cells are frozen on-the-fly. They change states.

Tail-recursive `map`

This code cannot be written in ML.

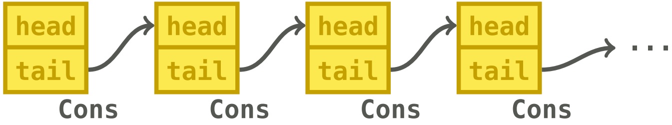


`Cons` blocks are immutable.
`Cell` blocks are mutable.

`Cons` cells are frozen on-the-fly. They change states.

Tail-recursive `map`

This code cannot be written in ML.



`Cons` blocks are immutable.
`Cell` blocks are mutable.

`Cons` cells are frozen on-the-fly. They change states.

Explaining the loop

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
    ): (| c0 @ list b)
  =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Cons
  | Cons { head = h; tail = t } ->
      let c1 = Cell { head = f h; tail = () } in
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Explaining the loop

Ownership

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
   ): (| c0 @ list b)
  =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Cons
  | Cons { head = h; tail = t } ->
      let c1 = Cell { head = f h; tail = () } in
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Function

```
map1 @ ... * f @ a -> b *
c0 @ Cell { head: b; tail: () } *
xs @ list a
```

Explaining the loop

Singleton

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
    ): (| c0 @ list b)
  =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Cons
  | Cons { head = h; tail = t } ->
      let c1 = Cell { head = f h; tail = () } in
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Structural

```
map1 @ ... * f @ a -> b *
c0 @ Cell { head: b; tail: =xs } *
xs @ Nil
```

Explaining the loop

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
    ): (| c0 @ list b)
  =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Cons
  | Cons { head = h; tail = t } ->
      let c1 = Cell { head = f h; tail = () } in
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Freeze

```
map1 @ ... * f @ a -> b *
c0 @ Cons { head: b; tail: =xs } *
xs @ Nil
```

Explaining the loop

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
    ): (| c0 @ list b)
  =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Cons
  | Cons { head = h; tail = t } ->
      let c1 = Cell { head = f h; tail = () } in
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Freeze

```
map1 @ ... * f @ a -> b *
c0 @ Cons { head: b; tail: Nil }
```

Explaining the loop

State change

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
    ): (| c0 @ list b)
  =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Cons
  | Cons { head = h; tail = t } ->
      let c1 = Cell { head = f h; tail = () } in
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Freeze

map1 @ ... * f @ a -> b *
c0 @ list b

Explaining the loop

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
    ): (| c0 @ li
  =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Cons
  | Cons { head = h; tail = t } ->
      let c1 = Cell { head = f h; tail = () } in
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Refine

```
map1 @ ... * f @ a -> b *
xs @ Cons { head: a; tail: list a } *
c0 @ Cell { head: b; tail: () }
```

Explaining the loop

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
    ): (| c0 @ li
  =
  match xs with
  | Nil ->
      c0.tail <-
      tag of c0 <- Cons
  | Cons { head = h; tail = t } ->
      let c1 = Cell { head = f h; tail = () } in
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Refine

```
map1 @ ... * f @ a -> b *
xs @ Cons { head: =h; tail: =t } *
h @ a * t @ list a *
c0 @ Cell { head: b; tail: () }
```

Explaining the loop

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
    ): (| c0 @ li
  =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Cons
  | Cons { head = h; tail = t } ->
      let c1 = Cell { head = f h; tail = () } in
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Refine

```
... *
h @ a * t @ list a *
c0 @ Cell { head: b; tail: () }
```

Explaining the loop

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
    ): (| c0 @ list b)
  =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Cons
  | Cons { head = h; tail = t } ->
      let c1 = Cell { head = f h; tail = () } in
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Assign

```
... *
h @ unknown * t @ list a *
c0 @ Cell { head: b; tail: =c1 } *
c1 @ Cell { head: b; tail: () } *
```

Explaining the loop

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
    ): (| c0 @ list b)
  =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Cons
  | Cons { head = h; ta
      let c1 = Cell { h
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Freeze

```
... *
h @ unknown * t @ list a *
c0 @ Cons { head: b; tail: =c1 } *
c1 @ Cell { head: b; tail: () } *
```

Explaining the loop

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
    ): (| c0 @ list b)
  =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Co
  | Cons { head = h;
      let c1 = Cell {
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Reasoning

```
... *
h @ unknown * t @ unknown *
c0 @ Cons { head: b; tail: =c1 } *
c1 @ list b
```

Explaining the loop

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
    ): (| c0 @ list b)
  =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Co
  | Cons { head = h;
      let c1 = Cell {
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Reasoning

```
... *
h @ unknown * t @ unknown *
c0 @ Cons { head: b; tail: list b }
```

Explaining the loop

```
val rec map1 [a, b] (
      f: a -> b,
      c0: Cell { head: b; tail: () },
      xs: list a
    ): (| c0 @ list b)
  =
  match xs with
  | Nil ->
      c0.tail <- xs;
      tag of c0 <- Co
  | Cons { head = h;
      let c1 = Cell {
      c0.tail <- c1;
      tag of c0 <- Cons;
      map1 (f, c1, t)
  end
```

Reasoning

```
... *
h @ unknown * t @ unknown *
c0 @ list b
```

How does it all work?

Thanks to...

- **singleton types** that encode equalities (~pure formulas) and allow rewriting,
- **structural types** that track the branch we are in,
- **folding** of inductive predicates,

...we manage to implement a very fine-grained reasoning within the type system.

Other interesting results

Other results that are not attainable in ML:

- in-place list reversal, while tracking ownership,
- `List.map` with sharing, while still having type
  ```
  val map: (list a, a -> b) -> list b
  ```
- in-place zipper (with ownership results), in-place tree traversal,
- iterators, with a precise ownership formulation.

Some of these are classical separation logic results.

Breaking out of the type system

Why?!

We're very happy with the type system but...

...aliasing on arbitrary, mutable data structures, cannot be expressed.

Two options

- extend the type system (complicated), or...
-

This is one of our key design choices.

Systems for reasoning statically exist; we want to explore a different tradeoff.

Two options

- extend the type system (complicated), or...
- rely on dynamic checks

This is one of our key design choices.

Systems for reasoning statically exist; we want to explore a different tradeoff.

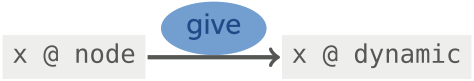
An example with complex ownership

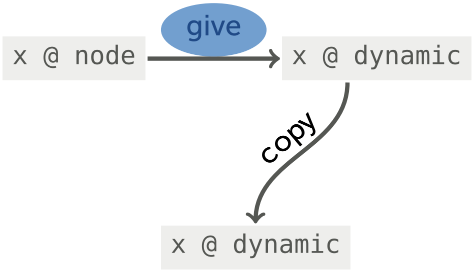
We need to represent a graph.

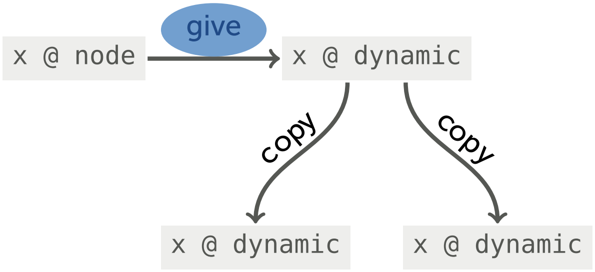
Imagine a DFS. We need to mark (mutable) nodes.

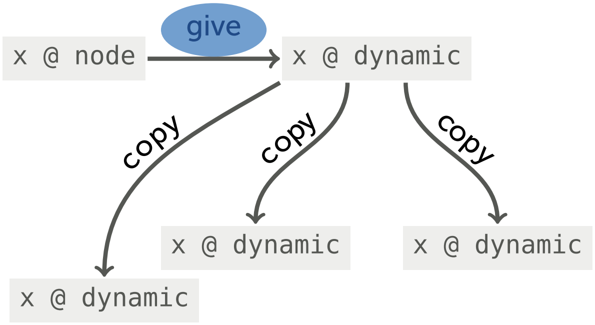
Multiple pointers to the same node. How do we guarantee the unique owner property for nodes?

```
x @ node
```

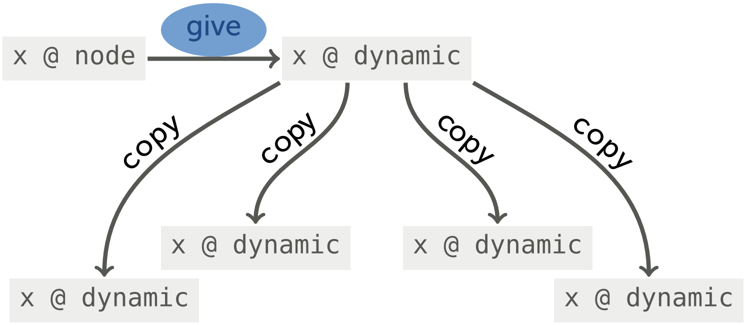

x @ node
give
x @ dynamic

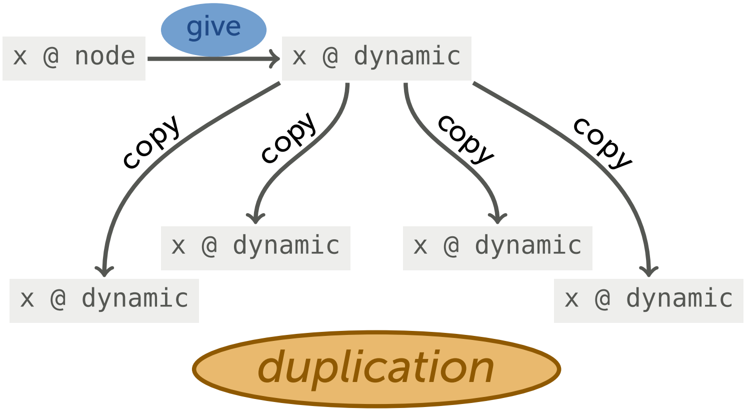

x @ node
give
x @ dynamic
copy
x @ dynamic

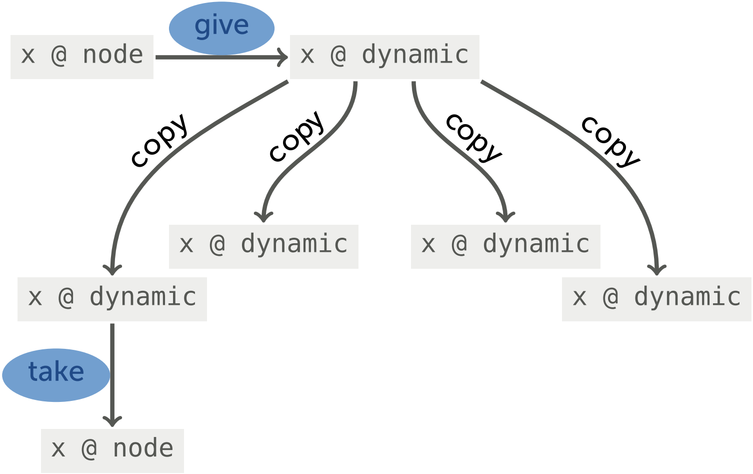

x @ node
give
x @ dynamic
copy
copy
x @ dynamic
x @ dynamic



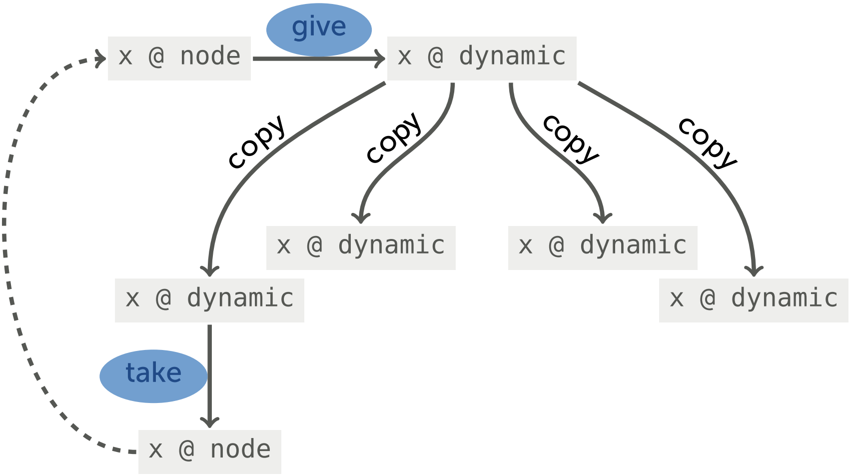
x @ node
give
x @ dynamic
copy
copy
copy
x @ dynamic
x @ dynamic
x @ dynamic

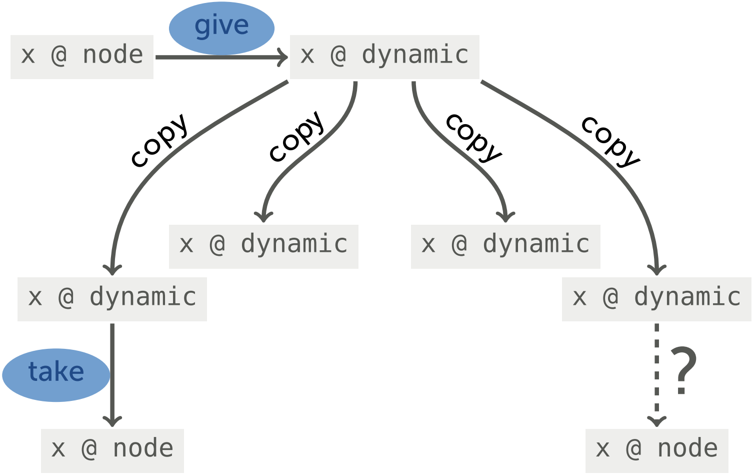

x @ node
give
x @ dynamic
copy
copy
copy
copy
x @ dynamic
x @ dynamic
x @ dynamic
x @ dynamic

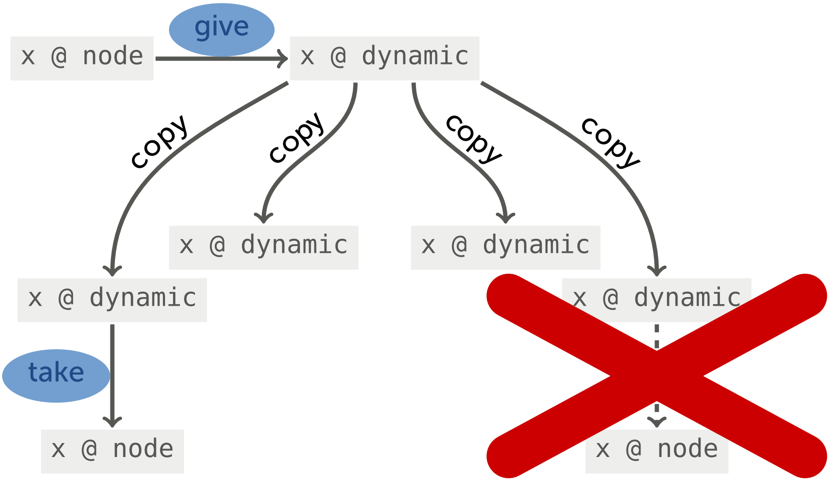

give
x @ node
x @ dynamic
copy
copy
copy
copy
x @ dynamic
x @ dynamic
x @ dynamic
x @ dynamic
duplication

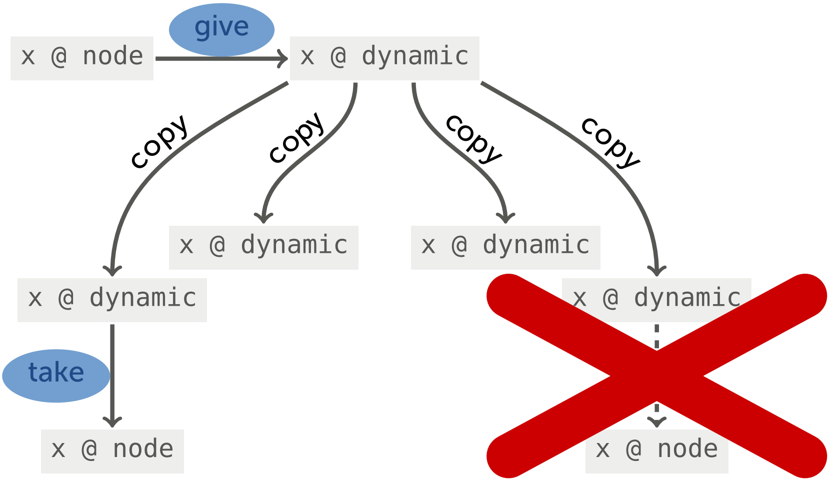


x @ node
give
x @ dynamic
copy
copy
copy
copy
x @ dynamic
x @ dynamic
x @ dynamic
x @ dynamic
take
x @ node


x @ node
give
x @ dynamic
copy
copy
copy
copy
x @ dynamic
x @ dynamic
x @ dynamic
x @ dynamic
take
x @ node


give
x @ node
x @ dynamic
copy
copy
copy
copy
x @ dynamic
x @ dynamic
x @ dynamic
x @ dynamic
take
?
x @ node
x @ node


give
x @ node
x @ dynamic
copy
copy
copy
copy
x @ dynamic
x @ dynamic
x @ dynamic
x @ dynamic
take
x @ node
x @ node



Uniqueness guaranteed *via* a runtime test

Under the hood

We have a notion of adopter and adoptee.

- Adopters declare the type of their adoptees.
- Adoptees maintain a pointer to their adopter telling whether they're “given” or “taken”.

We have a machine-checked proof of soundness (F. Pottier).

Advantages

- the adopter is exclusive: the `take` operation is lock-free;
- possible extension to duplicable adopters using `compare-and-swap`

From the programmer's point of view, a clear distinction between what is statically checked and what is not.

The state of Mezzo

Theory

- the type system of *Mezzo* is sound (F. Pottier)
- programs written in *Mezzo* are data-race free (T. Balabonski)

Implementation

- a type-checker has been written,
- requires type annotations in a few cases,
- connected to frame inference (separation logic) and join (shape analysis in abstract interpretation)

Living with *Mezzo*

Programming in *Mezzo*:

- forces the programmer to understand the ownership structure precisely,
- allows expressing strong invariants,
- allows new idioms (initialize-then-freeze).

It requires extra work from the programmer (error messages, type annotations). We believe the guarantees (data-race freedom, ownership properties) are worth the effort!

The final word

Mezzo: a programming language to talk about state, ownership and aliasing. The type system is sound. Programs written in *Mezzo* are data-race free.

New idioms, less bugs

Programming in Mezzo: come and see us at HOPE 2013 for a demo about iterators.

Learning about Mezzo: visit our website at http://protz.github.io/mezzo

How does it work? Adoption

An object can be declared as adopting other objects.

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

and mutable node a =
  Node  { children: list dynamic; payload: a }
```

How does it work? Adoption (cont'd)

```
(* x @ node a * f @ graph a *)
give x to f;
(* x @ dynamic * f @ graph a *)
```

`x @ dynamic` means "**`x`** *may* currently be adopted by some other object".

This is a duplicable permission.

How does it work? Abandon

We traded **`x @ cell a`** for **`x @ dynamic`**, which is duplicable but hides the true type of **`x`**.

```
(* x @ dynamic * f @ graph a *)
take x from f;
(* x @ node a * f @ graph a *)
```

We regain the original permission, but we need to make sure no object can be abandoned twice: **`abandon`** involves a dynamic check.

How does it work? Implementation

- Each object contains a hidden field with the address of its adopter, or **`null`**
- The field is set when adopting and cleared when abandoning.
- We perform the check when abandoning an object: its hidden field and the address of (what the user claims is) the adopter must match.

DFS (in surface syntax)

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