The design of *Mezzo*

François Pottier Jonathan Protzenko

INRIA

CMU, Sep 2013

Outline

- Motivation
- Design principles
- Algebraic data structures
- Extra examples
- Aliasing
- Project status

Premise

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.

Question

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;
- type changes along with state changes;
- (in certain cases) explicit memory re-use.

Outline

- Motivation
- Design principles
- Algebraic data structures
- Extra examples
- Aliasing
- Project status

Principle 1. Nothing is fixed

A variable x does not have a fixed type throughout its lifetime.

Instead,

- *at each program point* in the scope of x,
- one may have zero, one, or more (static) *permissions* to use x in certain ways.

Layout and ownership go hand in hand

As a consequence, permissions describe *layout* and *ownership*.

A permission of the form "`x @ t`" allows using `x` at type `t`.

It describes the *shape and extent* of a heap fragment, rooted at `x`, and describes certain *access rights* for this memory.

In short, "to know about `x`" is "to have access to `x`" is "to own `x`".

Principle 2. Just two access modes

The system imposes a global invariant: at any time,

- if x is a mutable object, there exists *at most one* permission to access it (for reading and writing);
- if x is an immutable object, there may exist arbitrarily *many* permissions to access it (for reading).

No counting. No fractions.

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 integer cells.

Principle 3. Any (known) alias is as good as any other

An equality “`x = y`” is a permission, sugar for “`x @ (=y)`”.

In its presence, “`x @ t`” can be turned into “`y @ t`”, and vice-versa.

No “borrowing”.

A value can be copied (always).

Can a permission be copied?

- “`x @ list int`” can be copied: read access can be shared.
- “`x = y`” can be copied: equalities are forever.
- “`x @ mlist int`” and “`x @ list (ref int)`” must not be copied, as they imply exclusive access to part of the heap.

One can always tell whether a permission is *duplicable* or *affine*.

```
let x = 0 in
let y = ref x in
let z = (y, y) in
...
```

We have "`x @ int`" and "`y @ ref (=x)`" and "`z @ (=y, =y)`".

Thus, we have "`x @ int`" and "`y @ ref int`" and "`z @ (=y, =y)`".

We *cannot* deduce "`z @ (ref int, ref int)`", as this reasoning step would require duplicating "`y @ ref int`".

Aliasing of mutable data is restricted.

```
let z : (ref int, ref int) = ... in
let (x, y) = z in
...
```

We have “`z @ (ref int, ref int)`” and “`z @ (=x, =y)`”.

I.e., “`z @ (=x, =y)`” and “`x @ ref int`” and “`y @ ref int`”.

We have an *exclusive* access token for each of `x` and `y`. There follows that these addresses *must be distinct*.

Technically, the word *“and”* above is a *conjunction* $*$ that requires *separation* at mutable data and *agreement* at immutable data.

Summary so far: the good

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.

Isn't this a restrictive discipline?

Yes, it is. In our defense,

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

Outline

- Motivation
- Design principles
- Algebraic data structures
- Extra examples
- Aliasing
- Project status

The algebraic data type of immutable lists is defined as in ML:

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

To define a type of mutable lists, one adds a keyword:

```
data mutable mlist a =
  | MNil
  | MCons { head: a; tail: mlist a }
```

```
match xs with
| MNil ->

    ...
| MCons ->

    let x = xs.head in

    ...
end
```

xs @ mlist a

```
match xs with
| MNil ->

    ...
| MCons ->

    let x = xs.head in

    ...
end
```

```
match xs with
| MNil ->

    ...
| MCons ->

    let x = xs.head in

    ...
end
```

xs @ MNil

```
match xs with
| MNil ->

    ...
| MCons ->

    let x = xs.head in

    ...
end
```

xs @ MCons { head: a; tail: mlist a }

```
match xs with
| MNil ->
    ...
| MCons ->

    let x = xs.head in

    ...
end
```

```
xs @ MCons { head: (=h); tail: (=t) }
* h @ a
* t @ mlist a
```

```
match xs with
| MNil ->

   ...
| MCons ->

   let x = xs.head in

   ...
end
```

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

```
match xs with
| MNil ->

    ...
| MCons ->

    let x = xs.head in

    ...
end
```

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

This illustrates two mechanisms:

- A nominal permission can be *unfolded* and *refined* to a structural one.
- A structural permission can be *decomposed* into a conjunction of permissions for the fields.

These reasoning steps are reversible.

This means that "`xs @ list (ref a)`" denotes a list of *pairwise distinct* references.

```
val length: [a] mlist a -> int
```

This type should be understood as follows:

- `length` requires one argument `xs`, along with the static permission "`xs @ mlist a`".
- `length` returns one result `n`, along with the static permission "`xs @ mlist a * n @ int`".

By convention, the permissions demanded by a function are also returned, unless the "`consumes`" keyword is used.

```
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)
```

xs @ 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)
```

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)
```

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)
```

xs @ 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)
```

xs @ MCons { head = h; tail = t }
h @ a
t @ mlist a

A recursive function

```
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)
```

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)
```

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)
```

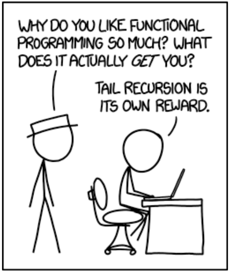
xs @ mlist a

Tail recursion versus iteration

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.



Outline

- Motivation
- Design principles
- Algebraic data structures
- **Extra examples**
- Aliasing
- Project status

A couple more examples:

- melding mutable lists;
- concatenating immutable lists.

Both feature iteration as tail recursion.

The latter also demonstrates *gradual initialization*.

Melding mutable lists (1/2)

```
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
```

not consumed!

```
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
```

consumed!

```
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)

```
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 = t }
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
```

```
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
```

xs @ MCons { head: a; tail: mlist a }
t @ MNil

```
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
```

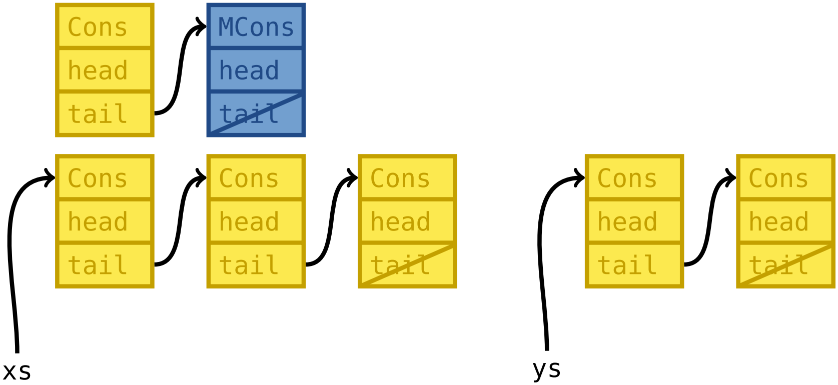
xs @ MCons { head: a; tail: mlist a }

```
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
```

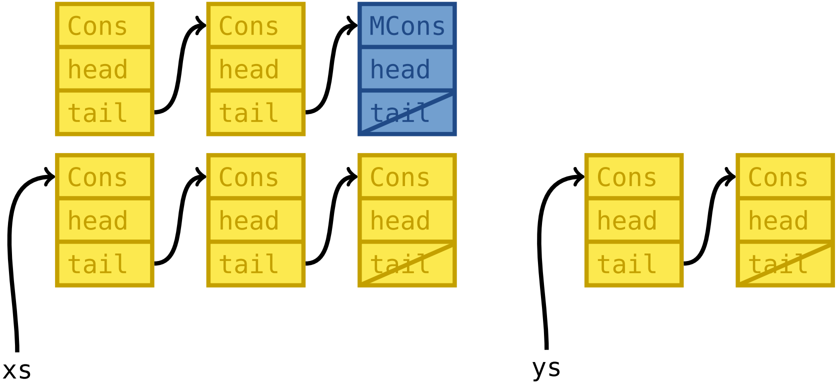
xs @ MCons { head: a; tail = t }
is framed out during the call

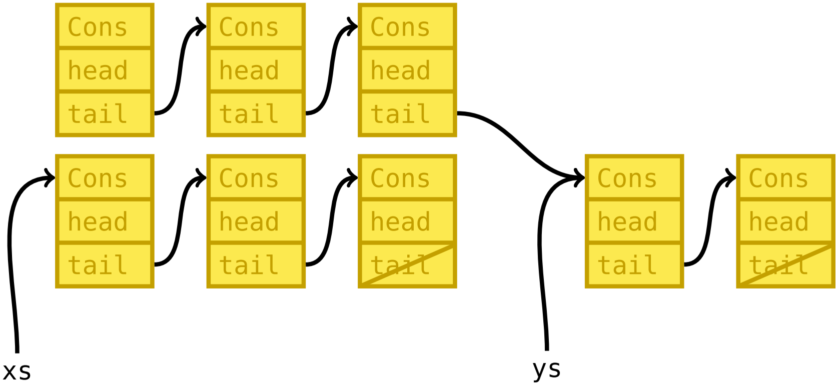
```
val meld [a] (consumes xs: mlist a,
              consumes ys: mlist a) : mlist a =
  match xs with
  | MNil  -> ys
  | MCons -> meld_aux (xs, ys); xs
  end
```

Concatenating immutable lists



Concatenating immutable lists




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
ys

We define a type for a partially-initialized “cons” cell:

```
alias mcons a =
    MCons { head: a; tail: () }
```

The permission “`c @ mcons a`” allows *updating* `c.tail`.

It also allows *freezing* the cell `c`.

```
val rec append_aux [a] (consumes (
     dst: mcons a, 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
```

Concatenating immutable lists (2/3)

```
val rec append_aux [a] (consumes (
     dst: mcons a, 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 consumed!

Concatenating immutable lists (2/3)

```
val rec append_aux [a] (consumes (
     dst: mcons a, 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 a list

Concatenating immutable lists (2/3)

```
val rec append_aux [a] (consumes (
     dst: mcons a, 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.tail is initialized

```
val rec append_aux [a] (consumes (
     dst: mcons a, 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 becomes an immutable block

Concatenating immutable lists (2/3)

```
val rec append_aux [a] (consumes (
     dst: mcons a, 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' becomes a valid list

```
val rec append_aux [a] (consumes (
     dst: mcons a, 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
```

hence, dst too becomes a valid list

```
val append [a] (consumes (xs: list a, ys: 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
```

Outline

- Motivation
- Design principles
- Algebraic data structures
- Extra examples
- Aliasing
- Project status

Aliasing is restricted

By default, mutable data cannot be aliased.

Several independent mechanisms circumvent this restriction:

- *locks* in the style of concurrent separation logic;
- *adoption and abandon*, an original feature;
- *nesting* in the style of Boyland.

The first two are more flexible, but are *runtime* mechanisms and can cause *deadlocks* and *runtime errors*.

We need two types:

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

abstract locked
```

The basic operations are:

```
val new:     [p: perm]
      (| consumes p)   -> lock p
val acquire: [p: perm]
      (l: lock p)      -> (| p * l @ locked)
val release: [p: perm]
      (l: lock p | consumes (p * l @ locked)) -> ()
```

"`try_acquire`" can also be expressed.

Locks are safe

In the presence of threads & locks,

- well-typed programs do not go wrong...
- ...and *are data-race-free* (Thibaut Balabonski, F.P.).

The type system does not prevent deadlocks.

Although "`x @ a`" cannot be shared, "`l @ lock (x @ a)`" can.

A value `x`, without any permission, can be shared too.

Thus, an object that is protected by a lock can be shared:

```
alias protected a =
  (x: unknown, lock (x @ a))
```

This allows an encoding of ML into *Mezzo*, of theoretical interest only, where every mutable object is protected by a lock.

Hiding a function's internal state allows sharing this function:

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

Hiding a function's internal state allows sharing this function:

```
val hide   [a, b, s : perm] (
  f : (a | s) -> b
| consumes s
)  : (a -> b) =
  let l : lock s = new () in
  fun (x : a) : b =
    acquire l;
    let y = f x in
    release l;
    y
```

Hiding a function's internal state allows sharing this function:

```
val hide   [a, b, s : perm] (
  f : (a | s) -> b
| consumes s
)  : (a -> b) =
  let l : lock s = new () in
  fun (x : a) : b =
    acquire l;
    let y = f x in
    release l;
    y
```

l @ lock s

Hiding a function's internal state allows sharing this function:

```
val hide   [a, b, s : perm] (
  f : (a | s) -> b
| consumes s
)  : (a -> b) =
  let l : lock s = new () in
  fun (x : a) : b =
    acquire l;
    let y = f x in
    release l;
    y
```

l @ lock s

Hiding a function's internal state allows sharing this function:

```
val hide   [a, b, s : perm] (
  f : (a | s) -> b
| consumes s
)  : (a -> b) =
  let l : lock s = new () in
  fun (x : a) : b =
    acquire l;
    let y = f x in
    release l;
    y
```

```
l @ locked
s
```

Adoption and abandon, also known as give & take, allow a single static permission to control a *group* of (mutable) objects.

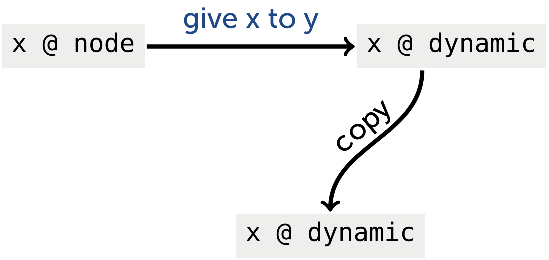
The objects in the group can be *aliased* in arbitrary ways.

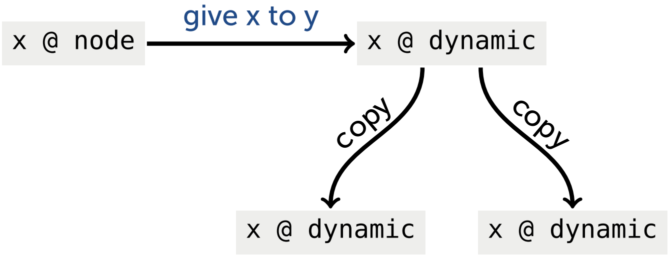
`x @ node`

give & take: overview

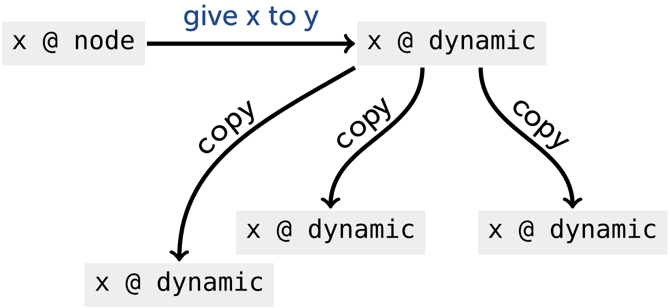
`x @ node` $\xrightarrow{\text{give x to y}}$ `x @ dynamic`

give & take: overview

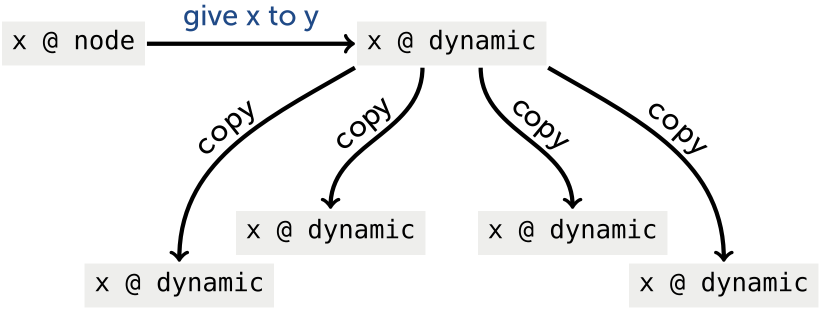



give x to y
x @ node
x @ dynamic
copy
copy
x @ dynamic
x @ dynamic

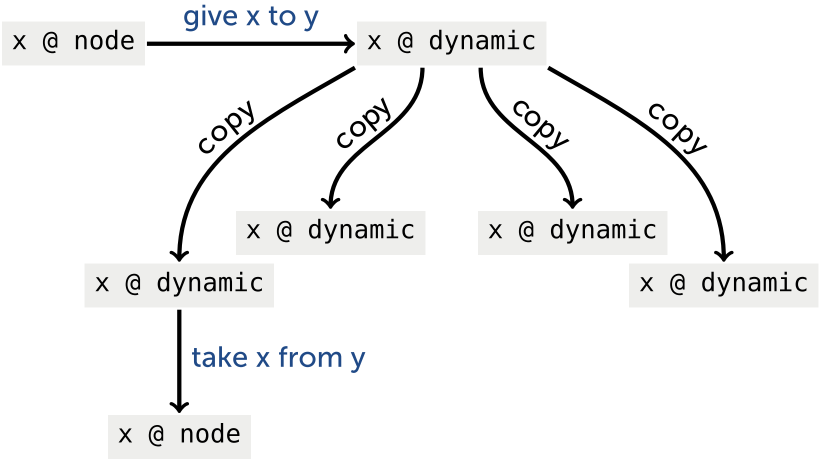
give & take: overview



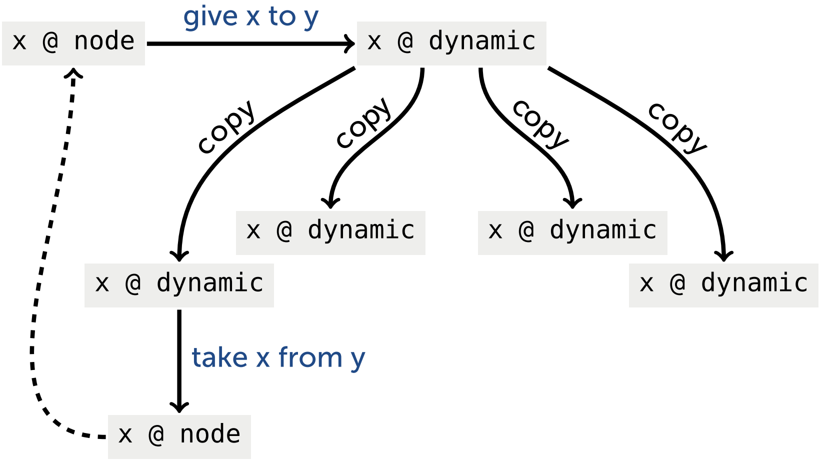
give & take: overview



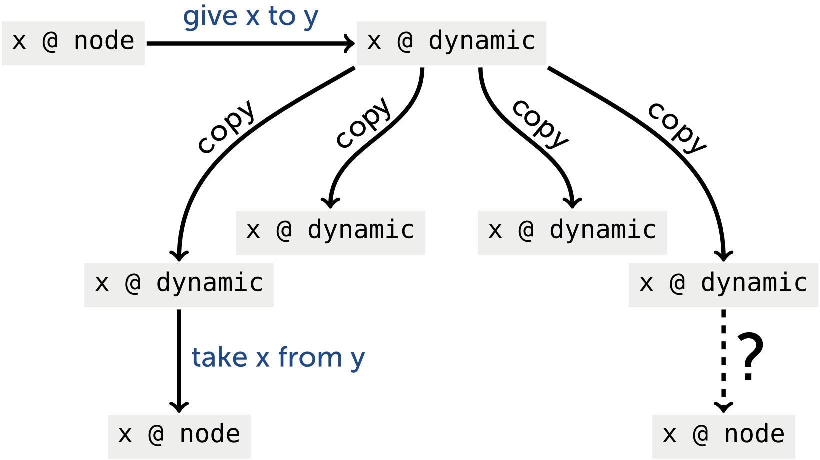
give & take: overview



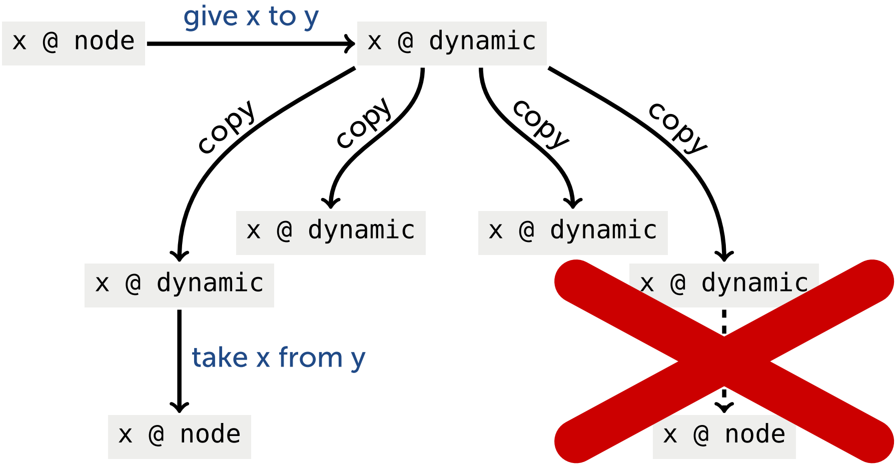
give & take: overview

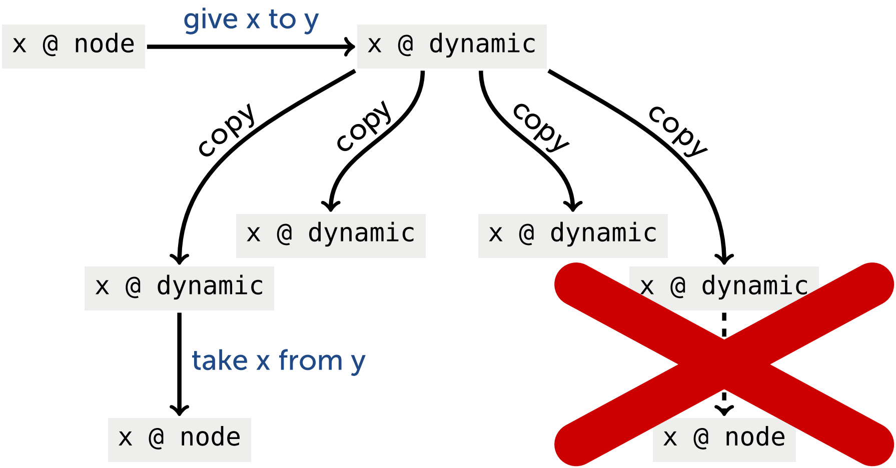


give & take: overview



give & take: overview





Uniqueness is guaranteed via a runtime check.

Mutable objects can serve as *adopters* or *adoptees*.

Every object maintains a (possibly `null`) pointer to its adopter.

"`give x to y`" sets this field; "`take x from y`" tests it and clears it.

"`take`" can *fail*.

“`give x to y`” and “`take x from y`” need exclusive ownership of `y`.

“`give x to y`” consumes “`x @ u`”, while “`take x from y`” produces “`x @ u`”, where the type `u` of the adoptee is determined by the type of the adopter.

Owning an object implicitly means owning all of its adoptees too.

give & take: bottom line

Well-typed programs do not go wrong, but can fail at "`take`".

This is a dynamic version of Fähndrich and DeLine's *regions* with adoption & focus.

The ownership hierarchy is *partly static, partly dynamic*.

Outline

- Motivation
- Design principles
- Algebraic data structures
- Extra examples
- Aliasing
- **Project status**

Who we are

The project was launched in late 2011 and currently involves

- *Jonathan Protzenko* (Ph.D student, soon to graduate),
- *Thibaut Balabonski* (post-doc researcher),
- and myself (INRIA researcher).

Where we are

We currently have:

- a formal definition and *type soundness proof* for Core *Mezzo*, including give & take and threads & locks;
- a prototype *type-checker*.

What next?

In the short term, we would like to:

- put more work into *type inference*, which is tricky;
- experimentally *evaluate* *Mezzo*'s expressiveness and usability;
- compile *Mezzo* down to untyped OCaml, or some other target.

What next?

Many as-yet-unanswered questions!

- Is this *a good mix* between static and dynamic mechanisms?
- Can we write *modular* code?
- Can we express *object protocols*?
- What about specifications & *proofs* of programs?

Thank you

More information is online at
http://gallium.inria.fr/~protzenk/mezzo-lang/