Mezzo

a typed language for safe and effectful concurrent programs

Jonathan Protzenko (INRIA)
jonathan.protzenko@inria.fr

This defense:

❶ some context;
❷ the design of *Mezzo*;
❸ 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.

```
# 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.

```
# 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.

```
# 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.

```
# 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?

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

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

$$\text{unit} \rightarrow \text{int}$$

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

$$\begin{array}{ll} \text{requires:} & r : \text{ref int} \\ \text{ensures:} & r : \text{ref int} \quad \wedge \\ & \text{old}(r.\text{contents}) + 1 = r.\text{contents} \quad \wedge \\ & \text{ret} = r.\text{contents} \end{array}$$

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:

$$(| r @ \mathsf{ref}\ \mathsf{int}) \rightarrow \mathsf{int}$$

Definitely not your run-of-the-mill ML type system, but not quite program proof either.

The *Mezzo* style of typing (2)

```
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 $*$.

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

In *Mezzo*: $P = x @ t * y @ u$

```
val f (x: ...): ... =
  let y = ... in
  ...
```

Mezzo is not ML

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

$$\text{In ML:}\quad \Gamma = x : t, y : u$$

$$\text{In } \textit{Mezzo}\text{:}\quad P = x \,@\, t * y \,@\, u$$

P_1

```
val f (x: ...): ... =
  let y = ... in
  ...
```

Mezzo is not ML

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

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

In *Mezzo*: $P = x @ t * y @ u$

P_2

```
val f (x: ...): ... =
  let y = ... in
  ...
```

Mezzo is not ML

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

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

In *Mezzo*: $P = x @ t * y @ u$

```
val f (x: ...): ... =
  let y = ... in
  ...
```

P_3

Different *modes* for types

	duplicable	*exclusive*
I	read-only	read-write
others	read-only	—

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.

Mezzo: a language with permissions

Function may consume ownership of their arguments.

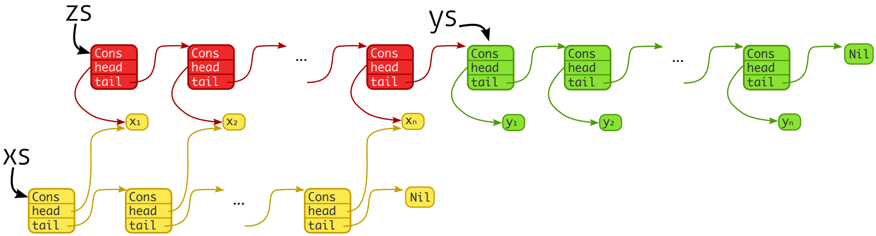
```
val append: [a] (
  consumes xs: list a,
  consumes ys: list a
) -> (zs: list a)
```

Mezzo: a language with permissions

Function may consume ownership of their arguments.

```
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:

```
iter_incr xs || iter_incr zs
```

How can we make this safe?

Mezzo: a language with permissions

Back to the signature.

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

Mezzo: a language with permissions

Example: **`list (ref int)`**.

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

Mezzo: a language with permissions

Exam

Before function call

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

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

Mezzo: a language with permissions

Example: **`list (ref int)`**.

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

After function call

`zs @ list (ref int)`

Mezzo: a language with permissions

Example: **`list int`**.

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

Mezzo: a language with permissions

Before function call

`xs @ list int * ys @ list int`

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

Mezzo: a language with permissions

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`**

Permissions

```
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] (
  consumes xs: list a,
  consumes ys: list a
): list a =
  match xs wit
  | Cons { hea  = 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
```

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
```

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
```

Permissions

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

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
```

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
```

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
```

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 @ 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
```

Permissions

```
ys @ list a *
xs @ Nil
```

```
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
```

Permissions

```
ret @ list a *
xs @ Nil *

ret = ys
```

The base layer

Mezzo is definitely not ML

Singleton types `x @ (=y)`: `x` is `y`
Written as: $x = y$

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

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

Several possible types

```
x @ (int, int),
x @ ∃(y,z: value).
  (=y | y @ int, =z | z @ int),
x @ ∃t.t, etc.
```

A glance at the type-checking rules

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

Sub

$$\frac{K; P_2 \vdash e : t_1 \qquad P_1 \leq P_2 \qquad t_1 \leq t_2}{K; P_1 \vdash e : t_2}$$

Frame

$$\frac{K; P_1 \vdash e : t}{K; P_1 * P_2 \vdash e : (t \mid P_2)}$$

Read

$$\frac{t \text{ is duplicable} \qquad P \text{ is } x \mathbin{@} \mathrm{A}\,\{\ldots; f : t; \ldots\}}{K; P \vdash x.f : (t \mid P)}$$

Tuple

$$K \vdash (x_1, \ldots, x_n) : (=x_1, \ldots, =x_n)$$

Application

$$K; x_1 \mathbin{@} t_2 \to t_1 * x_2 \mathbin{@} t_2 \vdash x_1\ x_2 : t_1$$

A glance at the subsumption relation

DecomposeTuple

$$y \,@\, (\ldots, t, \ldots) \equiv \exists (x : \mathsf{value})\ (y \,@\, (\ldots, {=}x, \ldots) * x \,@\, t)$$

EqualsForEquals

$$(y_1 = y_2) * [y_1/x]P \equiv (y_1 = y_2) * [y_2/x]P$$

EqualityReflexive

$$\mathsf{empty} \leq (x = x)$$

Fold

$$\frac{\mathsf{A}\ \{\vec{f} : \vec{t}\} \text{ is an unfolding of } X\ \vec{T}}{x \,@\, \mathsf{A}\ \{\vec{f} : \vec{t}\} \leq x \,@\, X\ \vec{T}}$$

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.

```
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) }
```

Initial permission

`n @ Node { neighbors: Nil; seen: bool }`

```
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) }
```

Fold

`n @ node`

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

data g
  roo
}

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) }
```

<u>Before Call</u>

```
n @ node * nil @ list node *
cons @ (consumes (node, list node)) -> list node
```

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

data gr
  roots
}

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) }
```

Function Call

~~n @ node * nil @ list node~~ *

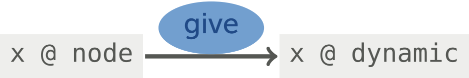
cons @ (consumes (node, list node)) -> list node

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

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

val g: graph =
  let n = Node { neighbors =                     in
  let neighbors = cons (n,
  n.neighbors <- neighbors;
  Graph { neighbors = cons (n, nil) }
```

Error

No field named neighbors for n.

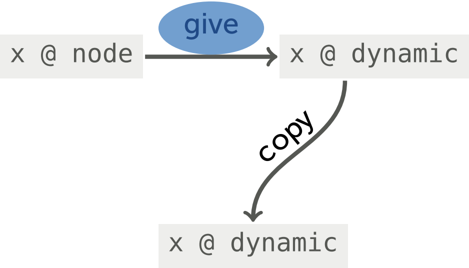
`x @ node`



`x @ node`

give

`x @ dynamic`



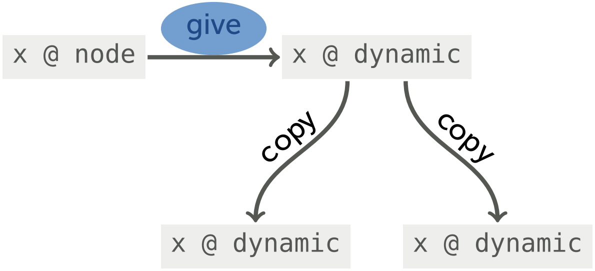
`x @ node`

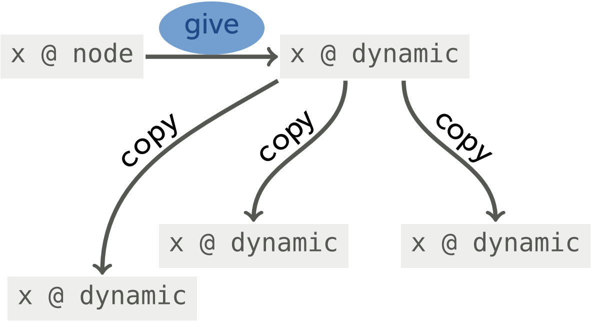
give

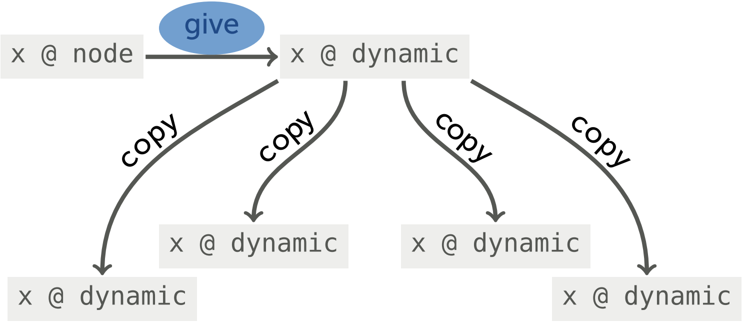
`x @ dynamic`

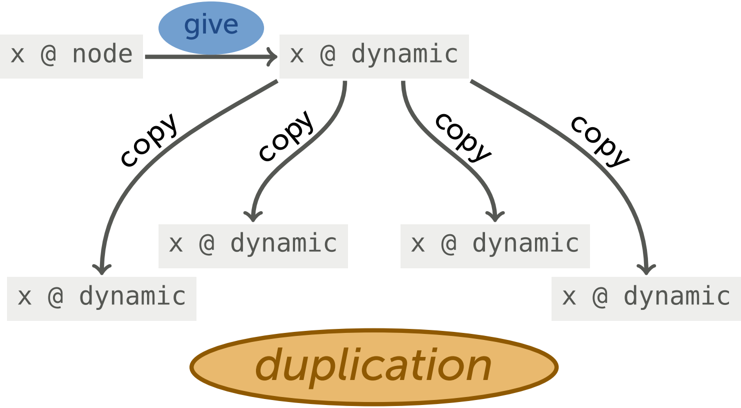
copy

`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



`x @ node`

give

`x @ dynamic`

copy

copy

copy

copy

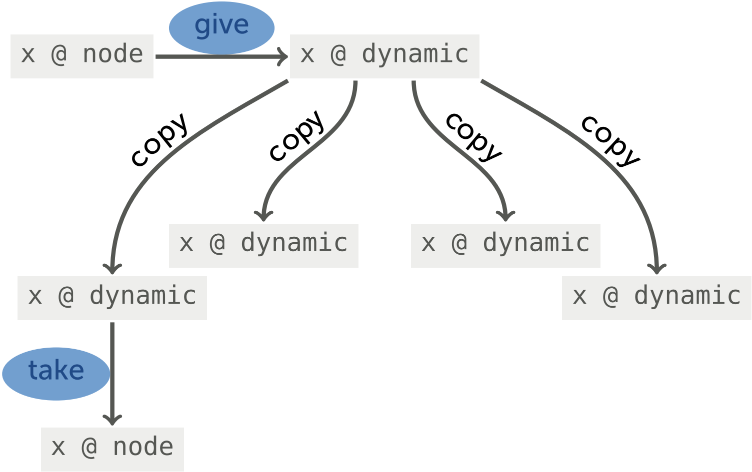
`x @ dynamic`

`x @ dynamic`

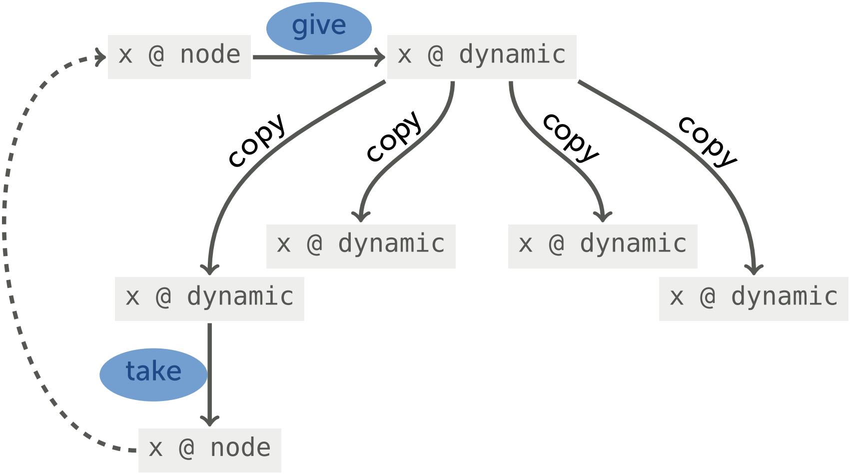
`x @ dynamic`

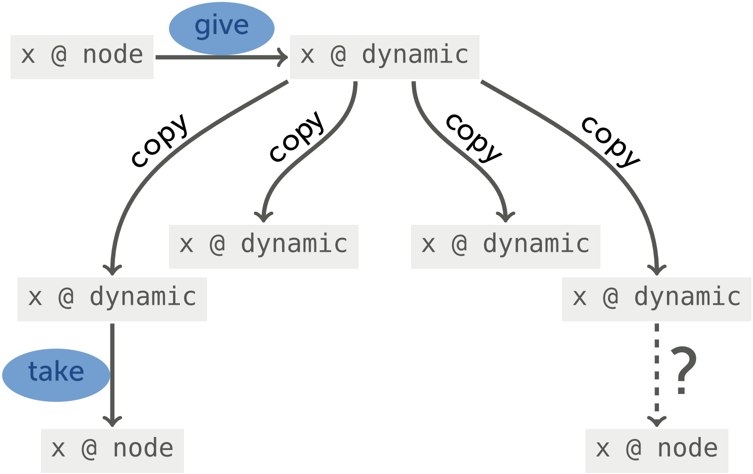
`x @ dynamic`

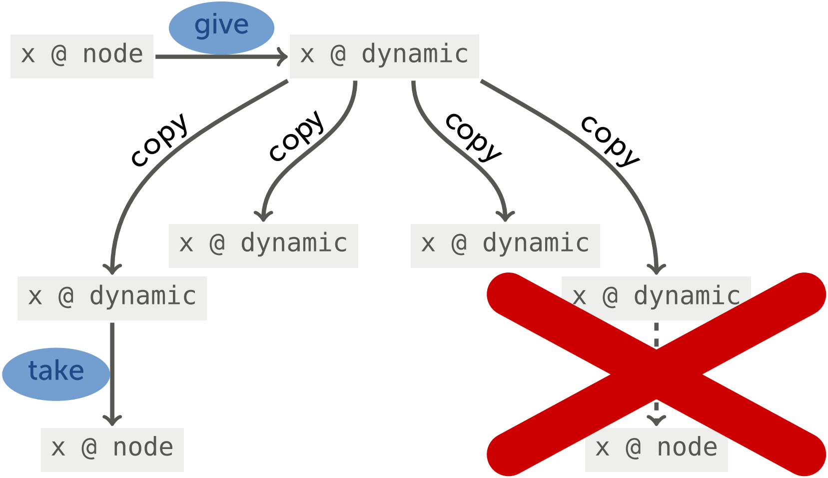
duplication



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



`x @ node`

give

`x @ dynamic`

copy

copy

copy

copy

`x @ dynamic`

`x @ dynamic`

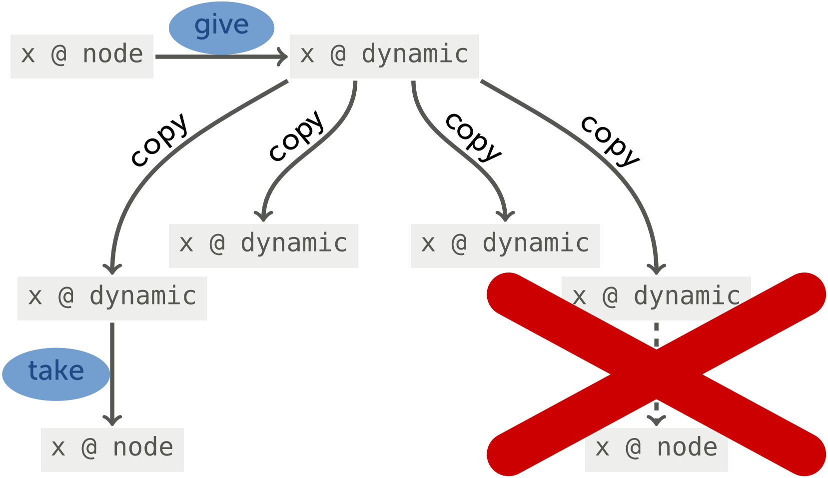
`x @ dynamic`

`x @ dynamic`

take

`x @ node`

`x @ node`



Uniqueness guaranteed *via* a runtime test

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

```
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

Permissions

```
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: ()
}
```

```
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: ()
} *

n @ 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
```

Permissions

```
n @ Node {
  contents: int; visited: bool;
  neighbors: ()
} *
n @ dynamic *
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
```

Permissions

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

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
} *
n @ dynamic *
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
```

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 = 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
} *
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 *
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 *

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`~~ `*`

`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~~ *`
`n @ dynamic *`
`ns @ list dynamic *`

`g @ graph`

A glance at the typing rules

x = adoptee, y = adopter

Give

$$\frac{t_2 \text{ adopts } t_1}{K; x \,@\, t_1 * y \,@\, t_2 \vdash \text{give } x \text{ to } y : (|\, y \,@\, t_2)}$$

Take

$$\frac{t_2 \text{ adopts } t_1}{K; x \,@\, \text{dynamic} * y \,@\, t_2 \vdash \text{take } x \text{ from } y : (|\, x \,@\, t_1 * y \,@\, t_2)}$$

Reflecting on the design of adoption/abandon

	run-time check	two-way	
static regions	✘	✘	
nesting	✘	✘	
locks	✓	✓	often used together
adoption/abandon	✓	✓	

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 `x₁ != x₂` 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

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

CopyDup

$$\frac{P \text{ is duplicable}}{C[t] * P \leq C[(t \mid P)] * P}$$

HideDuplicablePrecondition

$$\frac{P \text{ is duplicable}}{(x \,@\, (t_1 \mid P) \rightarrow t_2) * P \leq x \,@\, t_1 \rightarrow t_2}$$

ExistsIntro

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

CoArrow

$$\frac{u_1 \leq t_1 \qquad t_2 \leq u_2}{x \,@\, t_1 \rightarrow t_2 \leq x \,@\, u_1 \rightarrow u_2}$$

Unfold

$$\frac{\mathrm{A}\,\{\vec{f} : \vec{t}\} \text{ is an unfolding of } X\,\vec{T} \qquad X\,\vec{T} \text{ has only one branch}}{x \,@\, X\,\vec{T} \leq x \,@\, \mathrm{A}\,\{\vec{f} : \vec{t}\}}$$

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 @ (=x, =y) * x @ ref int * y @ ref int ≡`
  `z @ (ref int, ref int)`,
- inference (of type application): **`cons [?] (x, y)`**,
- subtyping:
  `[a] duplicable a => (ref a) -> a ≡`
  `[y: value] (ref (=y)) -> (=y)`,
- the frame rule...

Answer: normalization

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 as an asynchronous phase

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
- $f @ t \to u * x @ t \not\leq \exists (y : \text{value})\ y @ u$ (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

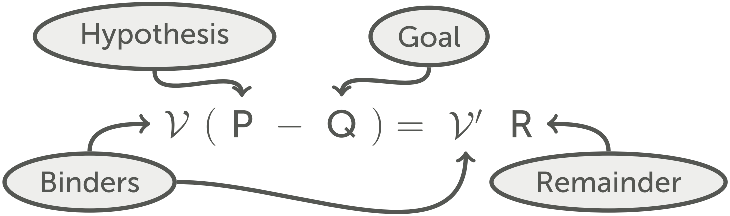
- A forward, flow-sensitive algorithm.
- Threads a normalized permission through program points.
- Relies on two algorithms: subtraction (deciding subtyping) and merge (simplifying disjunctions)

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 $P - Q = R$.



Hypothesis

Goal

$$\mathcal{V}\ (\ \mathrm{P}\ -\ \mathrm{Q}\ ) = \mathcal{V}'\ \ \mathrm{R}$$

Binders

Remainder

This means:

> "with the instantiation choices from $\mathcal{V}'$, we get $P \leq Q * R$".

Subtraction example

$\mathcal{R}$ denotes rigidly-bound variables.

$$
\begin{array}{l}
\quad \mathcal{R}(\ell, h, t). \\
\qquad \ell \mathbin{@} \mathsf{Cons}\,\{\mathsf{head} = h; \mathsf{tail} = t\} * \\
\qquad h \mathbin{@} \mathsf{ref\ int} * t \mathbin{@} \mathsf{list}\ (\mathsf{ref\ int}) \\
- \\
\qquad \ell \mathbin{@} \mathsf{list}\ (\mathsf{ref\ int}) \\
= \\
\quad \mathcal{R}(\ell, h, t). \\
\qquad \ell \mathbin{@} \mathsf{Cons}\,\{\mathsf{head} = h; \mathsf{tail} = t\}
\end{array}
$$

Backtracking

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

There may be several solutions:

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

Not all solutions are explored: α 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.

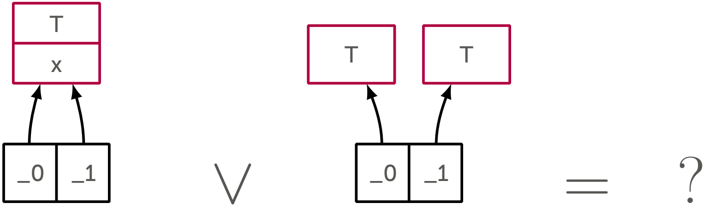
The implementation relies on:

- 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 = mutable T
```



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:

```
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:

```
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]. *)
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

```
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)

```
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
```