

# A right-to-left type system for mutually-recursive value definitions

Alban Reynaud, **Gabriel Scherer**, Jeremy Yallop

Parsifal, Inria Saclay, France

August 17, 2019



```
let rec fac = function
| 0 -> 1
| n -> n * fac (n - 1);;
(* val fac : int -> int = <fun> *)
fac 8;;
(* - : int = 40320 *)

let rec ones = 1 :: ones;;
(* val ones : int list = [1; <cycle>] *)
List.nth ones 10_000;;
(* - : int = 1 *)

let rec alot = 1 + alot;;
(* Error: This kind of expression is not allowed
   as right-hand side of 'let rec' *)
```

## Almost-killer app: toy interpreter

```
Adder := Fun(x): Fun(y): x+y
```

## Almost-killer app: toy interpreter

Adder := Fun(x): Fun(y): x+y

Adder(1)  $\rightarrow^*$  closure( $[x \mapsto 1]$ ,  $y \mapsto x + y$ )

## Almost-killer app: toy interpreter

Adder := Fun(x): Fun(y): x+y

Adder(1)  $\rightarrow^*$  closure( $[x \mapsto 1]$ ,  $y \mapsto x + y$ )

type ast = Var of var | ... | Fun of var \* expr

type value = ... | Closure of env \* var \* expr

and env = (var \* value) list

## Almost-killer app: toy interpreter

```
Adder := Fun(x): Fun(y): x+y
```

```
Adder(1) →* closure([x ↦ 1], y ↦ x + y)
```

```
type ast    = Var of var | ... | Fun of var * expr
type value  = ... | Closure of env * var * expr
and env     = (var * value) list
```

```
let rec eval env = function
| Var x -> List.assoc x env
| ...
| Fun (x, t) -> Closure(env, x, t)
```

## Almost-killer app: toy interpreter

Factorial := FunRec(f,n): if n=0 then 1 else n\*f(n-1)

Adder := Fun(x): Fun(y): x+y

Adder(1)  $\rightarrow^*$  closure([x  $\mapsto$  1], y  $\mapsto$  x + y)

type ast = Var of var | ... | Fun of var \* expr

type value = ... | Closure of env \* var \* expr

and env = (var \* value) list

let rec eval env = function

| Var x -> List.assoc x env

| ...

| Fun (x, t) -> Closure(env, x, t)

| FunRec (f, x, t) ->

## Almost-killer app: toy interpreter

Factorial := FunRec(f,n): if n=0 then 1 else n\*f(n-1)

Adder := Fun(x): Fun(y): x+y

Adder(1)  $\rightarrow^*$  closure([x  $\mapsto$  1], y  $\mapsto$  x + y)

type ast = Var of var | ... | Fun of var \* expr

type value = ... | Closure of env \* var \* expr

and env = (var \* value) list

let rec eval env = function

| Var x -> List.assoc x env

| ...

| Fun (x, t) -> Closure(env, x, t)

| FunRec (f, x, t) ->

(\* Closure((f, ?) :: env, x, t) \*)



## Almost-killer app: toy interpreter

Factorial := FunRec(f,n): if n=0 then 1 else n\*f(n-1)

Adder := Fun(x): Fun(y): x+y

Adder(1)  $\rightarrow^*$  closure([x  $\mapsto$  1], y  $\mapsto$  x + y)

type ast = Var of var | ... | Fun of var \* expr

type value = ... | Closure of env \* var \* expr

and env = (var \* value) list

```
let rec eval env = function
```

```
| Var x -> List.assoc x env
```

```
| ...
```

```
| Fun (x, t) -> Closure(env, x, t)
```

```
| FunRec (f, x, t) ->
```

```
  (* Closure((f, ?) :: env, x, t) *)
```

```
  let rec clo = Closure((f,clo) :: env, x, t) in clo
```

# State of the OCaml art

OCaml manual → Language Extensions → Recursive definitions of values

# State of the OCaml art

OCaml manual → Language Extensions → Recursive definitions of values

Complex syntactic description.

Not composable.

Hard to trust.

Did not age very well with new language features.

# State of the OCaml art

OCaml manual → Language Extensions → Recursive definitions of values

Complex syntactic description.

Not composable.

Hard to trust.

Did not age very well with new language features.

PR#7231: check too permissive with nested recursive bindings

PR#7215: Unsoundness with GADTs and let rec

PR#4989: Compiler rejects recursive definitions of values

PR#6939: Segfault with improper use of let-rec and float arrays

## State of the OCaml art

PR#7231: check too permissive with nested recursive bindings

```
let rec r = let rec x () = r
              and y () = x ()
            in y ()
          in r "oops"
```

## State of the OCaml art

PR#7215: Unsoundness with GADTs and let rec

```
let is_int (type a) : (int, a) eq =  
  let rec (p : (int, a) eq) =  
    match p with Refl -> Refl  
  in p
```

## State of the OCaml art

PR#4989: Compiler rejects recursive definitions of values

```
let rec f = let g = fun x -> f x in g
```

## State of the OCaml art

PR#6939: Segfault with improper use of let-rec and float arrays

```
let rec x = [| x |]; 1. in ()
```



## The typical approach

We propose a *type system* to check recursive value definitions.

Our types are one of five *access modes*  $m$ , with a typing judgment  $\Gamma \vdash t : m$ . A recursive declaration is safe if the mode of the recursive variables is gentle enough.

The typing rules are formulated so that an algorithm can easily be extracted.

We wrote the corresponding code; it landed in the OCaml compiler ([#556](#), April 2016; [#1942](#), July 2018), fixing more bugs than we introduced.

## Implementation

## Access modes

The mode of  $x$  in  $t$  is:

Ignore : 1

Delay :  $\lambda y. x$ , lazy  $x$ .

Guard :  $K(x)$

Return :  $x$ , let  $y = e$  in  $x$

Dereference :  $1 + x$ ,  $x y$ ,  $f x$ .

## Access modes

The mode of  $x$  in  $t$  is:

Ignore : 1

Delay :  $\lambda y. x$ , lazy  $x$ .

Guard :  $K(x)$

Return :  $x$ , let  $y = e$  in  $x$

Dereference :  $1 + x$ ,  $x y$ ,  $f x$ .

let rec  $f = \lambda n. n * f (n - 1)$

let rec  $o = \text{Cons}(1, o)$

let rec  $x = 1 + x$

let rec  $x = \text{let } y = x \text{ in } y$

$f : \text{Delay} \vdash \lambda n. n * f (n - 1) : \text{Return}$

$o : \text{Guard} \vdash \text{Cons}(1, o) : \text{Return}$

$x : \text{Dereference} \vdash 1 + x : \text{Return}$

$x : \text{Return} \vdash \text{let } y = x \text{ in } y : \text{Return}$

## Access modes

The mode of  $x$  in  $t$  is:

Ignore : 1

Delay :  $\lambda y. x$ , lazy  $x$ .

Guard :  $K(x)$

Return :  $x$ , let  $y = e$  in  $x$

Dereference :  $1 + x$ ,  $x y$ ,  $f x$ .

let rec  $f = \lambda n. n * f (n - 1)$

let rec  $o = \text{Cons}(1, o)$

let rec  $x = 1 + x$

let rec  $x = \text{let } y = x \text{ in } y$

$f : \text{Delay} \vdash \lambda n. n * f (n - 1) : \text{Return}$

$o : \text{Guard} \vdash \text{Cons}(1, o) : \text{Return}$

$x : \text{Dereference} \vdash 1 + x : \text{Return}$

$x : \text{Return} \vdash \text{let } y = x \text{ in } y : \text{Return}$

Safety criterion: recursive variables must have mode Guard or less.

## Mode typing judgment $\Gamma \vdash t : m$

Using  $t$  at mode Guard:  $K(t)$ .

Two readings of the judgment  $x : m_x \vdash t : m$ :

**left-to-right** : If  $x$  is safe at mode  $m_x$ , then  $t$  can be used at  $m$ .

**right-to-left** : Using  $t$  at  $m$  requires using  $x$  at  $m_x$ .

Right-to-left / backward reading:  $t, m$  inputs,  $\Gamma$  output

---

$$x : ? \quad \vdash \text{Pair}(1, \text{fst } x) : \text{Return}$$

## Mode typing judgment $\Gamma \vdash t : m$

Using  $t$  at mode Guard:  $K(t)$ .

Two readings of the judgment  $x : m_x \vdash t : m$ :

**left-to-right** : If  $x$  is safe at mode  $m_x$ , then  $t$  can be used at  $m$ .

**right-to-left** : Using  $t$  at  $m$  requires using  $x$  at  $m_x$ .

Right-to-left / backward reading:  $t, m$  inputs,  $\Gamma$  output

$$\frac{\overline{\emptyset \vdash 1 : \text{Guard}} \quad \overline{x : ? \quad \vdash \text{fst } x : \text{Guard}}}{x : ? \quad \vdash \text{Pair}(1, \text{fst } x) : \text{Return}}$$

## Mode typing judgment $\Gamma \vdash t : m$

Using  $t$  at mode Guard:  $K(t)$ .

Two readings of the judgment  $x : m_x \vdash t : m$ :

**left-to-right** : If  $x$  is safe at mode  $m_x$ , then  $t$  can be used at  $m$ .

**right-to-left** : Using  $t$  at  $m$  requires using  $x$  at  $m_x$ .

Right-to-left / backward reading:  $t, m$  inputs,  $\Gamma$  output

$$\frac{\frac{\frac{}{\emptyset \vdash 1 : \text{Guard}}}{x : ?} \quad \frac{\frac{x : ? \quad \vdash x : \text{Dereference}}{\vdash \text{fst } x : \text{Guard}}}{\vdash \text{Pair}(1, \text{fst } x) : \text{Return}}}{\vdash \text{Pair}(1, \text{fst } x) : \text{Return}}}{x : ?}$$

## Mode typing judgment $\Gamma \vdash t : m$

Using  $t$  at mode Guard:  $K(t)$ .

Two readings of the judgment  $x : m_x \vdash t : m$ :

**left-to-right** : If  $x$  is safe at mode  $m_x$ , then  $t$  can be used at  $m$ .

**right-to-left** : Using  $t$  at  $m$  requires using  $x$  at  $m_x$ .

Right-to-left / backward reading:  $t, m$  inputs,  $\Gamma$  output

$$\frac{\frac{}{\emptyset \vdash 1 : \text{Guard}}}{x : ?} \quad \frac{\frac{x : \text{Dereference} \vdash x : \text{Dereference}}{x : ? \quad \vdash \text{fst } x : \text{Guard}}}{\vdash \text{Pair}(1, \text{fst } x) : \text{Return}}}{\vdash \text{Pair}(1, \text{fst } x) : \text{Return}}$$



## Mode typing judgment $\Gamma \vdash t : m$

Using  $t$  at mode Guard:  $K(t)$ .

Two readings of the judgment  $x : m_x \vdash t : m$ :

**left-to-right** : If  $x$  is safe at mode  $m_x$ , then  $t$  can be used at  $m$ .

**right-to-left** : Using  $t$  at  $m$  requires using  $x$  at  $m_x$ .

Right-to-left / backward reading:  $t, m$  inputs,  $\Gamma$  output

$$\frac{\frac{}{\emptyset \vdash 1 : \text{Guard}}}{x : ?} \quad \frac{\frac{x : \text{Dereference} \vdash x : \text{Dereference}}{x : \text{Dereference} \vdash \text{fst } x : \text{Guard}}}{\vdash \text{Pair}(1, \text{fst } x) : \text{Return}}$$

## Mode typing judgment $\Gamma \vdash t : m$

Using  $t$  at mode Guard:  $K(t)$ .

Two readings of the judgment  $x : m_x \vdash t : m$ :

**left-to-right** : If  $x$  is safe at mode  $m_x$ , then  $t$  can be used at  $m$ .

**right-to-left** : Using  $t$  at  $m$  requires using  $x$  at  $m_x$ .

Right-to-left / backward reading:  $t, m$  inputs,  $\Gamma$  output

$$\frac{\frac{\emptyset \vdash 1 : \text{Guard}}{\quad} \quad \frac{x : \text{Dereference} \vdash x : \text{Dereference}}{x : \text{Dereference} \vdash \text{fst } x : \text{Guard}}}{x : \text{Dereference} \vdash \text{Pair}(1, \text{fst } x) : \text{Return}}$$

## Access modes algebra

The mode of  $x$  in  $C[x]$ : the mode action of the context  $C[\square]$ .

Ignore : 1

Delay :  $\lambda y. \square$ , lazy  $\square$ .

Guard :  $K(\square)$

Return :  $\square$ , let  $y = e$  in  $\square$

Dereference :  $1 + \square$ ,  $\square y, f \square$ .

## Access modes algebra

The mode of  $x$  in  $C[x]$ : the mode action of the context  $C[\square]$ .

Ignore : 1

Delay :  $\lambda y. \square$ , lazy  $\square$ .

Guard :  $K(\square)$

Return :  $\square$ , let  $y = e$  in  $\square$

Dereference :  $1 + \square$ ,  $\square y, f \square$ .

Total order: Ignore  $\prec$  Delay  $\prec$  Guard  $\prec$  Return  $\prec$  Dereference.

## Access modes algebra

The mode of  $x$  in  $C[x]$ : the mode action of the context  $C[\square]$ .

Ignore : 1

Delay :  $\lambda y. \square$ , lazy  $\square$ .

Guard :  $K(\square)$

Return :  $\square$ , let  $y = e$  in  $\square$

Dereference :  $1 + \square$ ,  $\square y, f \square$ .

Total order: Ignore  $\prec$  Delay  $\prec$  Guard  $\prec$  Return  $\prec$  Dereference.

Mode composition:  $C[C'[\square]]$  has mode action  $m[m']$ .

## Access modes algebra

The mode of  $x$  in  $C[x]$ : the mode action of the context  $C[\square]$ .

Ignore : 1

Delay :  $\lambda y. \square$ , lazy  $\square$ .

Guard :  $K(\square)$

Return :  $\square$ , let  $y = e$  in  $\square$

Dereference :  $1 + \square$ ,  $\square y, f \square$ .

Total order: Ignore  $\prec$  Delay  $\prec$  Guard  $\prec$  Return  $\prec$  Dereference.

Mode composition:  $C[C'[\square]]$  has mode action  $m[m']$ .

Ignore  $[m]$  = Ignore =  $m$  [Ignore]

Delay  $[m > \text{Ignore}]$  = Delay

Guard  $[\text{Return}]$  = Guard

Guard  $[m \neq \text{Return}]$  =  $m$

Return  $[m]$  =  $m$

Dereference  $[m > \text{Ignore}]$  = Dereference

Dereference  $[\text{Delay}] \neq \text{Delay} [\text{Dereference}]$   $f(\lambda x. \square), \lambda x. (f \square)$

## Access mode typing rules

$$\frac{}{\Gamma, x : m \vdash x : m} \qquad \frac{\Gamma \vdash t : m \quad m \succ m'}{\Gamma \vdash t : m'}$$

$$\frac{\Gamma, x : m_x \vdash t : m \text{ [Delay]}}{\Gamma \vdash \lambda x. t : m} \qquad \frac{\Gamma_t \vdash t : m \text{ [Dereference]} \quad \Gamma_u \vdash u : m \text{ [Dereference]}}{\Gamma_t + \Gamma_u \vdash t u : m}$$

$$\frac{(\Gamma_i \vdash t_i : m \text{ [Guard]})^i}{\sum (\Gamma_i)^i \vdash K(t_i)^i : m} \qquad \text{(pattern matching rules...)}$$

$$\frac{\Gamma_u, x : m_{x \in u} \vdash u : m}{? \quad \vdash \text{let rec } x = t \text{ in } u : m}$$

## Access mode typing rules

$$\frac{}{\Gamma, x : m \vdash x : m} \qquad \frac{\Gamma \vdash t : m \quad m \succ m'}{\Gamma \vdash t : m'}$$

$$\frac{\Gamma, x : m_x \vdash t : m \text{ [Delay]}}{\Gamma \vdash \lambda x. t : m} \qquad \frac{\Gamma_t \vdash t : m \text{ [Dereference]} \quad \Gamma_u \vdash u : m \text{ [Dereference]}}{\Gamma_t + \Gamma_u \vdash t u : m}$$

$$\frac{(\Gamma_i \vdash t_i : m \text{ [Guard]})^i}{\sum (\Gamma_i)^i \vdash K(t_i)^i : m} \qquad \text{(pattern matching rules...)}$$

$$\frac{\Gamma_t, x : m_{x \in t} \vdash t : \text{Return} \qquad \Gamma_u, x : m_{x \in u} \vdash u : m}{? \quad \vdash \text{let rec } x = t \text{ in } u : m}$$



## Access mode typing rules

$$\frac{}{\Gamma, x : m \vdash x : m} \qquad \frac{\Gamma \vdash t : m \quad m \succ m'}{\Gamma \vdash t : m'}$$

$$\frac{\Gamma, x : m_x \vdash t : m \text{ [Delay]}}{\Gamma \vdash \lambda x. t : m} \qquad \frac{\Gamma_t \vdash t : m \text{ [Dereference]} \quad \Gamma_u \vdash u : m \text{ [Dereference]}}{\Gamma_t + \Gamma_u \vdash t u : m}$$

$$\frac{(\Gamma_i \vdash t_i : m \text{ [Guard]})^i}{\sum (\Gamma_i)^i \vdash K(t_i)^i : m} \qquad \text{(pattern matching rules...)}$$

$$\frac{m_{x \in t} \leq \text{Guard} \quad \Gamma_t, x : m_{x \in t} \vdash t : \text{Return} \quad \Gamma_u, x : m_{x \in u} \vdash u : m}{? \quad \vdash \text{let rec } x = t \text{ in } u : m}$$

## Access mode typing rules

$$\frac{}{\Gamma, x : m \vdash x : m} \qquad \frac{\Gamma \vdash t : m \quad m \succ m'}{\Gamma \vdash t : m'}$$

$$\frac{\Gamma, x : m_x \vdash t : m \text{ [Delay]}}{\Gamma \vdash \lambda x. t : m} \qquad \frac{\Gamma_t \vdash t : m \text{ [Dereference]} \quad \Gamma_u \vdash u : m \text{ [Dereference]}}{\Gamma_t + \Gamma_u \vdash t u : m}$$

$$\frac{(\Gamma_i \vdash t_i : m \text{ [Guard]})^i}{\sum (\Gamma_i)^i \vdash K(t_i)^i : m} \qquad \text{(pattern matching rules...)}$$

$$\frac{m_{x \in t} \leq \text{Guard} \quad \Gamma_t, x : m_{x \in t} \vdash t : \text{Return} \quad \Gamma_u, x : m_{x \in u} \vdash u : m}{m_{x \in u} [\Gamma_t] + \Gamma_u \vdash \text{let rec } x = t \text{ in } u : m}$$

## Access mode typing rules

$$\frac{}{\Gamma, x : m \vdash x : m} \qquad \frac{\Gamma \vdash t : m \quad m \succ m'}{\Gamma \vdash t : m'}$$

$$\frac{\Gamma, x : m_x \vdash t : m \text{ [Delay]}}{\Gamma \vdash \lambda x. t : m} \qquad \frac{\Gamma_t \vdash t : m \text{ [Dereference]} \quad \Gamma_u \vdash u : m \text{ [Dereference]}}{\Gamma_t + \Gamma_u \vdash t u : m}$$

$$\frac{(\Gamma_i \vdash t_i : m \text{ [Guard]})^i}{\sum (\Gamma_i)^i \vdash K(t_i)^i : m} \qquad (\text{pattern matching rules...})$$

$$\frac{m_{x \in t} \leq \text{Guard} \quad \Gamma_t, x : m_{x \in t} \vdash t : \text{Return} \quad \Gamma_u, x : m_{x \in u} \vdash u : m}{m_{x \in u} [\Gamma_t] + \Gamma_u \vdash \text{let rec } x = t \text{ in } u : m}$$

## Access mode typing rules

$$\frac{}{\Gamma, x : m \vdash x : m} \qquad \frac{\Gamma \vdash t : m \quad m \succ m'}{\Gamma \vdash t : m'}$$

$$\frac{\Gamma, x : m_x \vdash t : m \text{ [Delay]}}{\Gamma \vdash \lambda x. t : m} \qquad \frac{\Gamma_t \vdash t : m \text{ [Dereference]} \quad \Gamma_u \vdash u : m \text{ [Dereference]}}{\Gamma_t + \Gamma_u \vdash t u : m}$$

$$\frac{(\Gamma_i \vdash t_i : m \text{ [Guard]})^i}{\sum (\Gamma_i)^i \vdash K(t_i)^i : m} \qquad \text{(pattern matching rules...)}$$

$$\frac{\Gamma_t, x : m_{x \in t} \vdash t : \text{Return} \quad m_{x \in t} \leq \text{Guard} \quad m'_{x \in u} \stackrel{\text{def}}{=} \max(m_{x \in u}, \text{Guard}) \quad \Gamma_u, x : m_{x \in u} \vdash u : m}{m'_{x \in u} [\Gamma_t] + \Gamma_u \vdash \text{let rec } x = t \text{ in } u : m}$$

## Soundness theorem

If  $\emptyset \vdash t : \text{Return}$   
and  $t \rightarrow^* t'$   
then  $t'$  is not going horribly wrong.

## Soundness theorem

If  $\emptyset \vdash t : \text{Return}$   
and  $t \rightarrow^* t'$   
then  $t'$  is not going horribly wrong.

What's a good operational semantics for `letrec`?

## Soundness theorem

If  $\emptyset \vdash t : \text{Return}$   
and  $t \rightarrow^* t'$   
then  $t'$  is not going horribly wrong.

What's a good operational semantics for `letrec`?

A source-level approach to `letrec`: explicit substitutions.

Hirschowitz, Leroy, and Wells (2003, 2009)

Nordlander, Carlsson, and Gill (2008)

$$\text{Vicious} \stackrel{\text{def}}{=} \{E_f[x] \mid \nexists v, (x = v) \stackrel{\text{ctx}}{\in} E_f\}$$

## Theorem

*If*

$$\emptyset \vdash t : \text{Return}$$

*and*

$$t \rightarrow^* t'$$

*then*

$$t' \notin \text{Vicious}$$

**Proof.**

Subject Reduction. □



## Related Work

**Backward analyses** We describe them as type systems. Syntax!

**Modal type theories** This is an instance of one – uni-typed.

**Modal type theories for (co)recursion** We have a nice inference algorithm.

**Degrees** Elaborate systems for objects and ML functors, need to accept more programs. Not uni-typed.

**Graphs as types** We don't.

**Operational semantics** Best order vs. worst order.

For more details, see our full paper:

<https://arxiv.org/abs/1811.08134>

# End.

- Tom Hirschowitz, Xavier Leroy, and J. B. Wells. [Compilation of extended recursion in call-by-value functional languages](#). In *PPDP*, 2003.
- Tom Hirschowitz, Xavier Leroy, and J. B. Wells. [Compilation of extended recursion in call-by-value functional languages](#). *Higher Order Symbol. Comput.*, 22(1), March 2009.
- Johan Nordlander, Magnus Carlsson, and Andy J. Gill. [Unrestricted pure call-by-value recursion](#). In *ML Workshop*, 2008.

## Bonus slide: reduction example

$$\text{match} \left( \begin{array}{l} \text{let rec } xs = \text{Cons}(1, xs) \text{ in} \\ \quad xs \end{array} \right) \text{ with} \left[ \begin{array}{l} \text{Nil} \quad \rightarrow \text{None} \\ \text{Cons}(y, ys) \rightarrow \text{Some}(ys) \end{array} \right]$$

→

## Bonus slide: reduction example

`match ( let rec xs = Cons (1, xs) in xs ) with [ Nil → None  
Cons (y, ys) → Some (ys)`

→

## Bonus slide: reduction example

$(xs = \text{Cons}(x, xs)) \in E[\square]$  (would work even if `let rec` at toplevel)

$\text{match} \left( \text{let rec } xs = \text{Cons}(1, xs) \text{ in } \underset{xs}{\square} \right) \text{ with } \left[ \begin{array}{l} \text{Nil} \quad \rightarrow \text{None} \\ \text{Cons}(y, ys) \rightarrow \text{Some}(ys) \end{array} \right]$

$\rightarrow \text{match} \left( \text{let rec } xs = \text{Cons}(1, xs) \text{ in } \underset{\text{Cons}(1, xs)}{\square} \right) \text{ with } \left[ \begin{array}{l} \text{Nil} \quad \rightarrow \text{None} \\ \text{Cons}(y, ys) \rightarrow \text{Some}(ys) \end{array} \right]$

## Bonus slide: reduction example

`match`  $\left( \begin{array}{l} \text{let rec } xs = \text{Cons}(1, xs) \text{ in} \\ \quad xs \end{array} \right)$  `with`  $\left[ \begin{array}{l} \text{Nil} \quad \rightarrow \text{None} \\ \text{Cons}(y, ys) \rightarrow \text{Some}(ys) \end{array} \right]$

$\rightarrow$  `match`  $\left( \begin{array}{l} \text{let rec } xs = \text{Cons}(1, xs) \text{ in} \\ \quad \text{Cons}(1, xs) \end{array} \right)$  `with`  $\left[ \begin{array}{l} \text{Nil} \quad \rightarrow \text{None} \\ \text{Cons}(y, ys) \rightarrow \text{Some}(ys) \end{array} \right]$

$\rightarrow$

## Bonus slide: reduction example

$(\text{let rec } (x_i = v_i)^i \text{ in } \dots)$

$\text{match} \left( \begin{array}{l} \text{let rec } xs = \text{Cons}(1, xs) \text{ in} \\ \quad xs \end{array} \right) \text{ with} \left[ \begin{array}{l} \text{Nil} \quad \rightarrow \text{None} \\ \text{Cons}(y, ys) \rightarrow \text{Some}(ys) \end{array} \right]$

$\rightarrow \text{match} \left( \begin{array}{l} \text{let rec } xs = \text{Cons}(1, xs) \text{ in} \\ \quad \text{Cons}(1, xs) \end{array} \right) \text{ with} \left[ \begin{array}{l} \text{Nil} \quad \rightarrow \text{None} \\ \text{Cons}(y, ys) \rightarrow \text{Some}(ys) \end{array} \right]$

$\rightarrow$

## Bonus slide: reduction example

$(\text{let rec } (x_i = v_i)^i \text{ in } \dots)$

$\text{match} \left( \begin{array}{l} \text{let rec } xs = \text{Cons}(1, xs) \text{ in} \\ xs \end{array} \right) \text{ with} \left[ \begin{array}{l} \text{Nil} \quad \rightarrow \text{None} \\ \text{Cons}(y, ys) \rightarrow \text{Some}(ys) \end{array} \right]$

$\rightarrow \text{match} \left( \begin{array}{l} \text{let rec } xs = \text{Cons}(1, xs) \text{ in} \\ \text{Cons}(1, xs) \end{array} \right) \text{ with} \left[ \begin{array}{l} \text{Nil} \quad \rightarrow \text{None} \\ \text{Cons}(y, ys) \rightarrow \text{Some}(ys) \end{array} \right]$

$\rightarrow \text{let rec } xs = \text{Cons}(1, xs) \text{ in}$



## Bonus slide: reduction example

`match ( let rec xs = Cons (1, xs) in  
          xs ) with [ Nil           →None  
                  Cons (y, ys)→Some (ys)`

→ `match ( let rec xs = Cons (1, xs) in  
          Cons (1, xs) ) with [ Nil           →None  
                              Cons (y, ys)→Some (ys)`

→ `let rec xs = Cons (1, xs) in`

## Bonus slide: reduction example

`match`  $\left( \begin{array}{l} \text{let rec } xs = \text{Cons}(1, xs) \text{ in} \\ \quad xs \end{array} \right)$  `with`  $\left[ \begin{array}{ll} \text{Nil} & \rightarrow \text{None} \\ \text{Cons}(y, ys) & \rightarrow \text{Some}(ys) \end{array} \right]$

$\rightarrow$  `match`  $\left( \begin{array}{l} \text{let rec } xs = \text{Cons}(1, xs) \text{ in} \\ \quad \text{Cons}(1, xs) \end{array} \right)$  `with`  $\left[ \begin{array}{ll} \text{Nil} & \rightarrow \text{None} \\ \text{Cons}(y, ys) & \rightarrow \text{Some}(ys) \end{array} \right]$

$\rightarrow$  `let rec`  $xs = \text{Cons}(1, xs)$  `in` `Some`  $(xs)$

## Bonus slide: Source term syntax

Terms  $\ni t, u ::= x, y, z$   
| let rec  $b$  in  $u$   
|  $\lambda x. t$  |  $t u$   
|  $K(t_i)^i$  | match  $t$  with  $h$

Bindings  $\ni b ::= (x_i = t_i)^i$   
Handlers  $\ni h ::= (p_i \rightarrow t_i)^i$   
Patterns  $\ni p, q ::= K(x_i)^i$

Values  $\ni v ::= \lambda x. t \mid K(w_i)^i \mid L[v]$

WeakValues  $\ni w ::= x \mid v \mid L[w]$

ValueBindings  $\ni B ::= (x_i = v_i)^i$

BindingCtx  $\ni L ::= \square \mid \text{let rec } B \text{ in } L$

Values $\ni v ::= \lambda x. t \mid K(w_i)^i \mid L[v]$	
WeakValues $\ni w ::= x \mid v \mid L[w]$	$F ::= \square t \mid t \square$
ValueBindings $\ni B ::= (x_i = v_i)^i$	$\mid K((t_i)^i, \square, (t_j)^j)$
BindingCtx $\ni L ::= \square \mid \text{let rec } B \text{ in } L$	$\mid \text{match } \square \text{ with } h$
	$\mid \text{let rec } b, x = \square, b' \text{ in } u$
EvalCtx $\ni E ::= \square \mid E[F]$	$\mid \text{let rec } B \text{ in } \square$
EvalFrame $\ni F$	

Values  $\ni v ::= \lambda x. t \mid K(w_i)^i \mid L[v]$   
 WeakValues  $\ni w ::= x \mid v \mid L[w]$   
 ValueBindings  $\ni B ::= (x_i = v_i)^i$   
 BindingCtx  $\ni L ::= \square \mid \text{let rec } B \text{ in } L$   
  
 EvalCtx  $\ni E ::= \square \mid E[F]$   
 EvalFrame  $\ni F$

$F ::= \square t \mid t \square$   
 $\mid K((t_i)^i, \square, (t_j)^j)$   
 $\mid \text{match } \square \text{ with } h$   
 $\mid \text{let rec } b, x = \square, b' \text{ in } u$   
 $\mid \text{let rec } B \text{ in } \square$

$$\frac{(x = v) \in^{\text{ctx}} E}{E[x] \rightarrow E[v]}$$

$$\frac{t \rightarrow^{\text{hd}} t'}{E[t] \rightarrow E[t']}$$

$$\begin{array}{l}
\text{Values } \ni v ::= \lambda x. t \mid K(w_i)^i \mid L[v] \\
\text{WeakValues } \ni w ::= x \mid v \mid L[w] \\
\text{ValueBindings } \ni B ::= (x_i = v_i)^i \\
\text{BindingCtx } \ni L ::= \square \mid \text{let rec } B \text{ in } L \\
\\
\text{EvalCtx } \ni E ::= \square \mid E[F] \\
\text{EvalFrame } \ni F ::= \square t \mid t \square \\
\quad \mid K((t_i)^i, \square, (t_j)^j) \\
\quad \mid \text{match } \square \text{ with } h \\
\quad \mid \text{let rec } b, x = \square, b' \text{ in } u \\
\quad \mid \text{let rec } B \text{ in } \square
\end{array}$$

$$\frac{(x = v) \stackrel{\text{ctx}}{\in} E}{E[x] \rightarrow E[v]} \quad \frac{t \rightarrow^{\text{hd}} t'}{E[t] \rightarrow E[t']} \quad \frac{(x = v) \stackrel{\text{frame}}{\in} F \quad \vee \quad (x = v) \stackrel{\text{ctx}}{\in} E}{(x = v) \stackrel{\text{ctx}}{\in} E[F]}$$

$$\frac{(x = v) \in B}{(x = v) \stackrel{\text{frame}}{\in} \text{let rec } B \text{ in } \square} \quad \frac{(x = v) \in (b \cup b')}{(x = v) \stackrel{\text{frame}}{\in} \text{let rec } b, y = \square, b' \text{ in } u}$$

$$\begin{array}{l}
\text{Values } \ni v ::= \lambda x. t \mid K(w_i)^i \mid L[v] \\
\text{WeakValues } \ni w ::= x \mid v \mid L[w] \\
\text{ValueBindings } \ni B ::= (x_i = v_i)^i \\
\text{BindingCtx } \ni L ::= \square \mid \text{let rec } B \text{ in } L \\
\\
\text{EvalCtx } \ni E ::= \square \mid E[F] \\
\text{EvalFrame } \ni F
\end{array}$$

$$\begin{array}{l}
F ::= \square t \mid t \square \\
\mid K((t_i)^i, \square, (t_j)^j) \\
\mid \text{match } \square \text{ with } h \\
\mid \text{let rec } b, x = \square, b' \text{ in } u \\
\mid \text{let rec } B \text{ in } \square
\end{array}$$

$$\frac{(x = v) \stackrel{\text{ctx}}{\in} E}{E[x] \rightarrow E[v]}$$

$$\frac{t \rightarrow^{\text{hd}} t'}{E[t] \rightarrow E[t']}$$

$$\frac{(x = v) \stackrel{\text{frame}}{\in} F \quad \vee \quad (x = v) \stackrel{\text{ctx}}{\in} E}{(x = v) \stackrel{\text{ctx}}{\in} E[F]}$$

$$\frac{(x = v) \in B}{(x = v) \stackrel{\text{frame}}{\in} \text{let rec } B \text{ in } \square}$$

$$\frac{(x = v) \in (b \cup b')}{(x = v) \stackrel{\text{frame}}{\in} \text{let rec } b, y = \square, b' \text{ in } u}$$

$$\frac{}{L[\lambda x. t] \quad v \rightarrow^{\text{hd}} L[t[v/x]]}$$

$$\frac{}{\text{match } L[K(w_i)^i] \text{ with } (\dots \mid K(x_i)^i \rightarrow u \mid \dots) \rightarrow^{\text{hd}} L[u[(w_i/x_i)^i]}}$$



$$\begin{aligned} \text{ForcingFrame} \ni F_f ::= & \square v \mid v \square \\ & \mid \text{match } \square \text{ with } h \\ & \mid \text{let rec } b, x = \square, b' \text{ in } t \\ \text{ForcingCtx} \ni E_f ::= & F_f \mid E[E_f] \mid E_f[L] \end{aligned}$$

$$\text{Vicious} \stackrel{\text{def}}{=} \{E_f[x] \mid \nexists v, (x = v) \stackrel{\text{ctx}}{\in} E_f\}$$

## Bonus slide: mutual recursion

$$\frac{(x_i : \Gamma_i)^i \vdash \text{rec } b \quad (m'_i)^i \stackrel{\text{def}}{=} (\max(m_i, \text{Guard}))^i \quad \Gamma_u, (x_i : m_i)^i \vdash u : m}{\sum (m'_i [\Gamma_i])^i + \Gamma_u \vdash \text{let rec } b \text{ in } u : m}$$

$$\frac{(\Gamma_i, (x_j : m_{i,j})^{j \in I} \vdash t_i : \text{Return})^{i \in I} \quad (m_{i,j} \preceq \text{Guard})^{i,j}}{(\Gamma'_i = \Gamma_i + \sum (m_{i,j} [\Gamma'_j])^j)^i \quad \Gamma'_i \vdash \text{rec } (x_i = t_i)^{i \in I}}$$