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

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

Parsifal, Inria Saclay, France

December 15, 2020

```
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
```

$$\texttt{Adder}(1) \rightarrow^{\star} \mathsf{closure}([x \mapsto 1],\ y \mapsto x + y)$$

# Almost-killer app: toy interpreter

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

$$\texttt{Adder}(1) \to^{\star} \text{closure}([x \mapsto 1],\ y \mapsto x + y)$$

```
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
```

$$\texttt{Adder}(1) \to^{\star} \text{closure}([x \mapsto 1],\ y \mapsto x + y)$$

```
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)
```

# 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
```

$$\texttt{Adder}(1) \to^{\star} \text{closure}([x \mapsto 1],\ y \mapsto x + y)$$

```
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
```

$$\texttt{Adder}(1) \to^{\star} \text{closure}([x \mapsto 1],\ y \mapsto x + y)$$

```
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
```

$$\texttt{Adder}(1) \rightarrow^{\star} \mathsf{closure}([x \mapsto 1],\; y \mapsto x + y)$$

```
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 $\rightarrow$ Language Extensions $\rightarrow$ 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 : $x$, $\texttt{let}\ y = e\ \texttt{in}\ x$
- Dereference : $1 + x$, $x\ y$, $f\ x$.

# Access modes

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

$$\begin{aligned}\text{Ignore} &: 1 \\ \text{Delay} &: \lambda y.\, x, \text{ lazy } x. \\ \text{Guard} &: K\,(x) \\ \text{Return} &: x, \texttt{let}\ y = e\ \texttt{in}\ x \\ \text{Dereference} &: 1 + x,\ x\ y,\ f\ x.\end{aligned}$$

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

# Access modes

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

| | |
|---|---|
| Ignore | : $1$ |
| Delay | : $\lambda y.\, x$, lazy $x$. |
| Guard | : $K\,(x)$ |
| Return | : $x$, $\texttt{let}\ y = e\ \texttt{in}\ x$ |
| Dereference | : $1 + x$, $x\ y$, $f\ x$. |

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

| | |
|---|---|
| $\texttt{let rec}\ f = \lambda n.\, n * f\ (n-1)$ | $f$ : Delay $\vdash \lambda n.\, n * f\ (n-1)$ : Return |
| $\texttt{let rec}\ o = \texttt{Cons}(1, o)$ | $o$ : Guard $\vdash$ $Cons\,(1, o)$ : Return |
| $\texttt{let rec}\ x = 1 + x$ | $x$ : Dereference $\vdash 1 + x$ : Return |
| $\texttt{let rec}\ x = \texttt{let}\ y = x\ \texttt{in}\ y$ | $x$ : Return $\vdash \texttt{let}\ y = x\ \texttt{in}\ y$ : Return |

# Access modes

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

| | |
|---|---|
| Ignore | : $1$ |
| Delay | : $\lambda y.\, x$, lazy $x$. |
| Guard | : $K\,(x)$ |
| Return | : $x$, $\texttt{let}\ y = e\ \texttt{in}\ x$ |
| Dereference | : $1 + x$, $x\ y$, $f\ x$. |

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

| | |
|---|---|
| $\texttt{let rec}\ f = \lambda n.\, n * f\ (n-1)$ | $f : \text{Delay} \vdash \lambda n.\, n * f\ (n-1) : \text{Return}$ |
| $\texttt{let rec}\ o = \texttt{Cons}(1, o)$ | $o : \text{Guard} \vdash Cons\,(1, o) : \text{Return}$ |
| $\texttt{let rec}\ x = 1 + x$ | $x : \text{Dereference} \vdash 1 + x : \text{Return}$ |
| $\texttt{let rec}\ x = \texttt{let}\ y = x\ \texttt{in}\ y$ | $x : \text{Return} \vdash \texttt{let}\ y = x\ \texttt{in}\ y : \text{Return}$ |

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

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

# 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

$$\dfrac{\dfrac{}{\emptyset \vdash 1 : \text{Guard}} \qquad \dfrac{}{x : ? \qquad \vdash \texttt{fst}\ x : \text{Guard}}}{x : ? \qquad \vdash \texttt{Pair}\,(1, \texttt{fst}\ x) : \text{Return}}$$

# 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

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

# 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

$$
\dfrac{\dfrac{}{\emptyset \vdash 1 : \text{Guard}} \qquad \dfrac{\dfrac{}{x : \text{Dereference} \vdash x : \text{Dereference}}}{x : ? \qquad\qquad \vdash \texttt{fst}\ x : \text{Guard}}}{x : ? \qquad\qquad \vdash \texttt{Pair}\,(1, \texttt{fst}\ x) : \text{Return}}
$$

# 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

$$\dfrac{\dfrac{}{\emptyset \vdash 1 : \text{Guard}} \qquad \dfrac{\dfrac{}{x : \text{Dereference} \vdash x : \text{Dereference}}}{x : \text{Dereference} \vdash \texttt{fst}\ x : \text{Guard}}}{x : ? \qquad\qquad \vdash \texttt{Pair}\,(1, \texttt{fst}\ x) : \text{Return}}$$

# 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

$$\dfrac{\dfrac{}{\emptyset \vdash 1 : \mathsf{Guard}} \qquad \dfrac{\dfrac{}{x : \mathsf{Dereference} \vdash x : \mathsf{Dereference}}}{x : \mathsf{Dereference} \vdash \mathtt{fst}\ x : \mathsf{Guard}}}{x : \mathsf{Dereference} \vdash \mathtt{Pair}\,(1, \mathtt{fst}\ x) : \mathsf{Return}}$$

# 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 : $\square$, $\texttt{let}\ y = e\ \texttt{in}\ \square$

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

# Access modes algebra

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

$$
\begin{aligned}
\text{Ignore} &: 1 \\
\text{Delay} &: \lambda y.\,\square,\ \text{lazy}\ \square. \\
\text{Guard} &: K\,(\square) \\
\text{Return} &: \square,\ \texttt{let}\ y = e\ \texttt{in}\ \square \\
\text{Dereference} &: 1 + \square,\ \square\ y,\ f\ \square.
\end{aligned}
$$

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 : $\square$, $\texttt{let}\ y = e\ \texttt{in}\ \square$

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

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

$$
\begin{array}{lclcl}
\text{Ignore}\,[m] & = & \text{Ignore} & = & m\,[\text{Ignore}] \\
\text{Delay}\,[m > \text{Ignore}] & = & \text{Delay} & & \\
\text{Guard}\,[\text{Return}] & = & \text{Guard} & & \\
\text{Guard}\,[m \neq \text{Return}] & = & m & & \\
\text{Return}\,[m] & = & m & & \\
\text{Dereference}\,[m > \text{Ignore}] & = & \text{Dereference} & &
\end{array}
$$

Dereference [Delay] $\neq$ Delay [Dereference] $\qquad f\ (\lambda x.\square), \lambda x.(f\ \square)$

# Access mode typing rules

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

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

$$\frac{(\Gamma_i \vdash t_i : m\,[\text{Guard}])^i}{\sum (\Gamma_i)^i \vdash K\,(t_i)^i : m}$$

(pattern matching rules...)

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

# Access mode typing rules

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

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

$$\frac{(\Gamma_i \vdash t_i : m\,[\text{Guard}])^i}{\sum (\Gamma_i)^i \vdash K\,(t_i)^i : m}$$

(pattern matching rules...)

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

# Access mode typing rules

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

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

$$\frac{\left(\Gamma_i \vdash t_i : m\,[\text{Guard}]\right)^i}{\sum (\Gamma_i)^i \vdash K\,(t_i)^i : m}$$

(pattern matching rules...)

$$\frac{\begin{array}{c} m_{x\in t} \leq \text{Guard} \\ \Gamma_t, x : m_{x\in t} \vdash t : \text{Return} \end{array} \qquad \Gamma_u, x : m_{x\in u} \vdash u : m}{? \qquad \vdash \texttt{let rec}\ x = t\ \texttt{in}\ u : m}$$

# Access mode typing rules

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

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

$$\frac{\left(\Gamma_i \vdash t_i : m\,[\mathrm{Guard}]\right)^i}{\sum \left(\Gamma_i\right)^i \vdash K\,\left(t_i\right)^i : m}$$

(pattern matching rules...)

$$\frac{\begin{array}{c} m_{x\in t} \leq \mathrm{Guard} \\ \Gamma_t, x : m_{x\in t} \vdash t : \mathrm{Return} \end{array} \qquad \Gamma_u, x : m_{x\in u} \vdash u : m}{m_{x\in u}\,[\Gamma_t] + \Gamma_u \vdash \texttt{let rec } x = t \texttt{ in } u : m}$$

# Access mode typing rules

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

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

$$\frac{(\Gamma_i \vdash t_i : m\,[\mathrm{Guard}])^i}{\sum (\Gamma_i)^i \vdash K\,(t_i)^i : m}$$

(pattern matching rules...)

$$\frac{\begin{array}{c} m_{x \in t} \leq \mathrm{Guard} \\ \Gamma_t, x : m_{x \in t} \vdash t : \mathrm{Return} \end{array} \qquad \Gamma_u, x : m_{x \in u} \vdash u : m}{m_{x \in u}[\Gamma_t] + \Gamma_u \vdash \texttt{let rec } x = t \texttt{ in } u : m}$$

# Access mode typing rules

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

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

$$\frac{\left(\Gamma_i \vdash t_i : m\,[\text{Guard}]\right)^i}{\sum \left(\Gamma_i\right)^i \vdash K\,(t_i)^i : m}$$

(pattern matching rules...)

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

# Soundness theorem

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

# Soundness theorem

If $\emptyset \vdash t : \mathrm{Return}$
and $t \to^{\star} t'$
then $t'$ is not going horribly wrong.

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

# Soundness theorem

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

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

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

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

**Theorem**

*If*

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

*and*

$$t \to^\star t'$$

*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.

# Bonus slide: reduction example

$$\texttt{match}\left(\begin{array}{l}\texttt{let rec}\ xs = \texttt{Cons}\,(1, xs)\ \texttt{in} \\ \qquad xs\end{array}\right)\texttt{with}\left[\begin{array}{ll}\texttt{Nil} & \to \texttt{None} \\ \texttt{Cons}\,(y, ys) & \to \texttt{Some}\,(ys)\end{array}\right.$$

$\rightarrow$

# Bonus slide: reduction example

$$\texttt{match}\ \left(\begin{array}{l}\texttt{let rec}\ \mathit{xs} = \texttt{Cons}\,(1, \mathit{xs})\ \texttt{in} \\ \qquad \mathit{xs}\end{array}\right)\ \texttt{with}\ \left[\begin{array}{ll}\texttt{Nil} & \to \texttt{None} \\ \texttt{Cons}\,(y, \mathit{ys}) & \to \texttt{Some}\,(\mathit{ys})\end{array}\right.$$

$\to$

# Bonus slide: reduction example

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

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

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

# Bonus slide: reduction example

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

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

$$\to$$

# Bonus slide: reduction example

$$(\texttt{let rec } (x_i = v_i)^i \texttt{ in } \ldots)$$

$$\texttt{match} \begin{pmatrix} \texttt{let rec } xs = \texttt{Cons}\,(1, xs) \texttt{ in} \\ xs \end{pmatrix} \texttt{with} \left[ \begin{array}{l} \texttt{Nil} \to \texttt{None} \\ \texttt{Cons}\,(y, ys) \to \texttt{Some}\,(ys) \end{array} \right.$$

$$\to \texttt{match} \begin{pmatrix} \texttt{let rec } xs = \texttt{Cons}\,(1, xs) \texttt{ in} \\ \texttt{Cons}\,(1, xs) \end{pmatrix} \texttt{with} \left[ \begin{array}{l} \texttt{Nil} \to \texttt{None} \\ \texttt{Cons}\,(y, ys) \to \texttt{Some}\,(ys) \end{array} \right.$$

$$\to$$

# Bonus slide: reduction example

$$(\texttt{let rec } (x_i = v_i)^i \texttt{ in } \ldots)$$

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

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

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

# Bonus slide: reduction example

$$\texttt{match}\left(\begin{array}{l}\texttt{let rec } xs = \texttt{Cons}\,(1, xs)\texttt{ in}\\ \qquad xs\end{array}\right)\texttt{with}\left[\begin{array}{ll}\texttt{Nil} & \to \texttt{None}\\ \texttt{Cons}\,(y, ys) & \to \texttt{Some}\,(ys)\end{array}\right.$$

$$\to\ \texttt{match}\left(\begin{array}{l}\texttt{let rec } xs = \texttt{Cons}\,(1, xs)\texttt{ in}\\ \qquad \texttt{Cons}\,(1, xs)\end{array}\right)\texttt{with}\left[\begin{array}{ll}\texttt{Nil} & \to \texttt{None}\\ \texttt{Cons}\,(y, ys) & \to \texttt{Some}\,(ys)\end{array}\right.$$

$$\to\ \texttt{let rec } xs = \texttt{Cons}\,(1, xs)\texttt{ in}$$

# Bonus slide: reduction example

$$\texttt{match}\left(\begin{array}{l}\texttt{let rec}\ xs = \texttt{Cons}\,(1, xs)\ \texttt{in} \\ \qquad xs\end{array}\right)\texttt{with}\left[\begin{array}{ll}\texttt{Nil} & \to \texttt{None} \\ \texttt{Cons}\,(y, ys) & \to \texttt{Some}\,(ys)\end{array}\right.$$

$$\to\quad \texttt{match}\left(\begin{array}{l}\texttt{let rec}\ xs = \texttt{Cons}\,(1, xs)\ \texttt{in} \\ \qquad \texttt{Cons}\,(1, xs)\end{array}\right)\texttt{with}\left[\begin{array}{ll}\texttt{Nil} & \to \texttt{None} \\ \texttt{Cons}\,(y, ys) & \to \texttt{Some}\,(ys)\end{array}\right.$$

$$\to\quad \texttt{let rec}\ xs = \texttt{Cons}\,(1, xs)\ \texttt{in}\ \ \texttt{Some}\,(xs)$$

# Bonus slide: Source term syntax

$$
\begin{aligned}
\text{Terms} \ni t, u ::= {} & x, y, z \\
\mid {} & \texttt{let rec } b \texttt{ in } u \\
\mid {} & \lambda x.\, t \mid t\ u \\
\mid {} & K\,(t_i)^i \mid \texttt{match } t \texttt{ with } h
\end{aligned}
$$

$$
\begin{aligned}
\text{Bindings} \ni b \quad & ::= (x_i = t_i)^i \\
\text{Handlers} \ni h \quad & ::= (p_i \to t_i)^i \\
\text{Patterns} \ni p, q & ::= K\,(x_i)^i
\end{aligned}
$$

$$
\begin{aligned}
\text{Values} \ni v &::= \lambda x.\, t \mid K\,(w_i)^i \mid L[v] \\
\text{WeakValues} \ni w &::= x \mid v \mid L[w] \\
\text{ValueBindings} \ni B &::= (x_i = v_i)^i \\
\text{BindingCtx} \ni L &::= \Box \mid \texttt{let rec}\ B\ \texttt{in}\ L
\end{aligned}
$$

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

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

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

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

$$
\frac{(x = v) \overset{\text{ctx}}{\in} E}{E[x] \to E[v]} \qquad \frac{t \to^{\text{hd}} t'}{E[t] \to E[t']}
$$

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

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

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

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

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

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

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

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

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

$$
\frac{}{\texttt{match}\ L[K\,\overline{(w_i)}^{i}]\ \texttt{with}\ (\ldots \mid K\,\overline{(x_i)}^{i} \to u \mid \ldots) \to^{\text{hd}} L[u[\overline{(w_i/x_i)}^{i}]]}
$$

$$\begin{aligned}
\mathsf{ForcingFrame} \ni F_{\mathsf{f}} &::= \Box\ v \mid v\ \Box \\
&\mid \texttt{match}\ \Box\ \texttt{with}\ h \\
&\mid \texttt{let rec}\ b, x = \Box, b'\ \texttt{in}\ t \\
\mathsf{ForcingCtx} \ni E_{\mathsf{f}} &::= F_{\mathsf{f}} \mid E[E_{\mathsf{f}}] \mid E_{\mathsf{f}}[L]
\end{aligned}$$

$$\mathsf{Vicious} \stackrel{\mathsf{def}}{=} \{E_{\mathsf{f}}[x] \mid \nexists v, (x = v) \stackrel{\mathsf{ctx}}{\in} E_{\mathsf{f}}\}$$

# Bonus slide: mutual recursion

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

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