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

\[ \text{Adder}(1) \rightarrow^* \text{closure}\left([x \mapsto 1, \ y \mapsto x + y]\right) \]
Almost-killer app: toy interpreter

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

Adder(1) \rightarrow^* \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

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

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

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

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

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) →* closure([x ↦ 1], y ↦ x + y)

type ast = Var of var | ... | Fun of 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

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

\[
\text{type ast} = \text{Var of var} | \ldots | \text{Fun of var * expr} \\
\text{type value} = \ldots | \text{Closure of env * var * expr} \\
\text{and env} = (\text{var * value}) \text{ list}
\]

let rec eval env = function
| Var x -> List.assoc x env
| \ldots
| 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

```ocaml
let rec r = let rec x () = r
    and y () = x ()
    in y ()
    in r "oops"
```
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**: $\lambda y \cdot x$, lazy $x$.
- **Guard**: $K(x)$
- **Return**: $x$, let $y = e$ in $x$
- **Dereference**: $1 + x$, $x \ y$, $f \ x$.

Safety criterion: recursive variables must have mode Guard or less.
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**: $1 + x$, $x y$, $f x$.

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

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

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

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

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

\[
\text{let rec } f = \lambda n. n \times f (n - 1) \quad f : \text{Delay} \vdash \lambda n. n \times f (n - 1) : \text{Return}
\]
\[
\text{let rec } o = \text{Cons}(1, o) \quad o : \text{Guard} \vdash \text{Cons}(1, o) : \text{Return}
\]
\[
\text{let rec } x = 1 + x \quad x : \text{Dereference} \vdash 1 + x : \text{Return}
\]
\[
\text{let rec } x = \text{let } y = x \text{ in } y \quad x : \text{Return} \vdash \text{let } y = x \text{ in } y : \text{Return}
\]

**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

\[ x : ? \vdash \text{Pair}(1, \text{fst } x) : \text{Return} \]
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

\[
\begin{align*}
\emptyset \vdash 1 : \text{Guard} & \quad x : ? \quad \vdash \text{fst } x : \text{Guard} \\
& \quad x : ? \quad \vdash \text{Pair}(1, \text{fst } x) : \text{Return}
\end{align*}
\]
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

\[
\begin{array}{c}
\emptyset \vdash 1 : \text{Guard} \\
x : ? \quad \vdash x : \text{Dereference} \\
\hline
x : ? \quad \vdash \text{Pair}(1, \text{fst } x) : \text{Return}
\end{array}
\]
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

\[
\begin{array}{ccc}
\emptyset \vdash 1 : \text{Guard} & & x : \text{Dereference} \vdash x : \text{Dereference} \\
\hline
x : ? & & x : ? \vdash \text{fst } x : \text{Guard} \\
\hline
& & \vdash \text{Pair}(1, \text{fst } x) : \text{Return}
\end{array}
\]
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

\[
\begin{array}{c}
\emptyset \vdash 1 : \text{Guard} \\
x : \text{Dereference} \vdash x : \text{Dereference} \\
\hline
x : ? \\
\hline
\vdash \text{Pair}(1, \text{fst } x) : \text{Return}
\end{array}
\]
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

\[
\begin{align*}
\emptyset \vdash 1 : \text{Guard} & \quad x : \text{Dereference} \vdash x : \text{Dereference} \\
\hline
x : \text{Dereference} \vdash \text{Pair}(1, \text{fst } x) : \text{Return} & \quad x : \text{Dereference} \vdash \text{fst } x : \text{Guard}
\end{align*}
\]
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**: \( \lambda y. \square, \text{ lazy } \square \).
- **Guard**: \( K (\square) \)
- **Return**: \( \square, \text{ let } y = e \text{ in } \square \)
- **Dereference**: \( 1 + \square, \square y, f \square \).

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 $C[\Box]$.

- **Ignore** : $1$
- **Delay** : $\lambda y. \Box$, lazy $\Box$.
- **Guard** : $K(\Box)$
- **Return** : $\Box$, let $y = e$ in $\Box$

**Dereference** : $1 + \Box$, $\Box y$, $f \Box$.

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

| Expression              | Mode Action |
|-------------------------|-------------|
| 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 |

$\text{Dereference \ [Delay]} \neq \text{Delay \ [Dereference]}$

$f (\lambda x. \Box), \lambda x. (f \Box)$
Access mode typing rules

\[
\text{Γ}, x : m \vdash x : m
\]

\[
\frac{\text{Γ}, x : m \vdash t : m}{\text{Γ} \vdash \lambda x. t : m}
\]

\[
\frac{\text{Γ}, x : m \vdash t : m \text{ [Delay]}}{\text{Γ} \vdash \lambda x. t : m}
\]

\[
\frac{\text{Γ} \vdash t : m \text{ [Dereference]}}{\text{Γ} \vdash \lambda x. t : m}
\]

\[
\frac{\text{Γ} \vdash u : m \text{ [Dereference]}}{\text{Γ} + \text{Γ} \vdash t \ u : m}
\]

\[
\frac{\sum (\text{Γ}_i) \vdash K(t_i)^i : m}{\text{Γ} \vdash \text{let rec } x = t \text{ in } u : m}
\]

\[
\frac{(\text{Γ}_i \vdash t_i : m \text{ [Guard]})^i}{? \vdash \text{let rec } x = t \text{ in } u : m}
\]

(pattern matching rules...)
Access mode typing rules

\[
\begin{align*}
\Gamma, x : m & \vdash x : m \\
\Gamma & \vdash \lambda x. t : m
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash t : m \\
& \quad m \succ m'
\end{align*}
\]

\[
\Gamma, x : m_x & \vdash t : m \quad \text{[Delay]} \\
\Gamma & \vdash \lambda x. t : m
\]

\[
\begin{align*}
\Gamma & \vdash t : m \quad \text{[Dereference]} \\
\Gamma u & \vdash u : m \quad \text{[Dereference]}
\end{align*}
\]

\[
\Gamma t + \Gamma u \vdash t \; u : m
\]

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

\[
\begin{align*}
\Gamma t, x : m_x \in t & \vdash t : \text{Return} \\
\Gamma u, x : m_x \in u & \vdash u : m
\end{align*}
\]

\[
\frac{? \vdash \text{let rec } x = t \text{ in } u : m}{\quad \vdash \text{let rec } x = t \text{ in } u : m}
\]
Access mode typing rules

\[
\Gamma, x : m \vdash x : m \\
\frac{\Gamma \vdash t : m \quad m \succ m'}{\Gamma \vdash t : m'}
\]

\[
\Gamma, x : m_x \vdash t : m \text{ [Delay]} \\
\frac{\Gamma \vdash \lambda x. t : m}{\Gamma \vdash \lambda x. t : m}
\]

\[
\frac{\Gamma_t \vdash t : m \text{ [Dereference]} \quad \Gamma_u \vdash u : m \text{ [Dereference]}}{\Gamma_t + \Gamma_u \vdash t \ u : m}
\]

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

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

\[
\Gamma_u, x : m_{x \in u} \vdash u : m
\]
Access mode typing rules

\[
\frac{\Gamma, x : m \vdash x : m}{\Gamma' \vdash t : m}
\]

\[
\frac{\Gamma \vdash t : m \quad m > m'}{\Gamma \vdash t : m'}
\]

\[
\frac{\Gamma, x : m_x \vdash t : m}{\Gamma \vdash \lambda x. t : m}
\]

\[
\frac{\Gamma_t \vdash t : m \quad \Gamma_u \vdash u : m}{\Gamma_t + \Gamma_u \vdash t \ u : m}
\]

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

\[
\frac{m_{x \in t} \leq \text{Guard}}{\Gamma_t, x : m_{x \in t} \vdash t : \text{Return}}
\]

\[
\frac{\Gamma_u, x : m_{x \in u} \vdash u : m}{m_{x \in u} [\Gamma_t] + \Gamma_u \vdash \text{let rec } x = t \text{ in } u : m}
\]
Access mode typing rules

\[
\frac{}{\Gamma, x : m \vdash x : m}
\]

\[
\frac{}{\Gamma \vdash t : m \quad m \succ m'}
\]

\[
\frac{}{\Gamma \vdash t : m'}
\]

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

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

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

\[
\frac{}{(\Gamma_i)^i \vdash K(t_i)^i : m}
\]

\[
\frac{}{(\text{pattern matching rules...})}
\]

\[
\frac{}{m_{x \in t} \leq \text{Guard}}
\]

\[
\frac{}{\Gamma_t, x : m_{x \in t} \vdash t : \text{Return}}
\]

\[
\frac{}{\Gamma_u, x : m_{x \in u} \vdash u : m}
\]

\[
\frac{}{m_{x \in u}[\Gamma_t] + \Gamma_u \vdash \text{let \ rec \ } x = t \ in \ u : m}
\]
Access mode typing rules

\[
\frac{\Gamma, x : m \vdash x : m}{\Gamma, x : m \vdash t : m} \quad \frac{\Gamma \vdash t : m \quad m \succ m'}{\Gamma \vdash t : m'}
\]

\[
\Gamma, x : m_x \vdash t : m \quad \text{[Delay]} \quad \Gamma \vdash \lambda x. t : m
\]

\[
\Gamma \vdash t : m \quad \text{[Dereference]} \quad \Gamma_u \vdash u : m \quad \text{[Dereference]}
\]

\[
\Gamma_t + \Gamma_u \vdash t \ u : m
\]

\[
\sum (\Gamma_i)^i \vdash K (t_i)^i : m
\]

\[
\left(\Gamma_i \vdash t_i : m \ \text{[Guard]}\right)^i
\]

\[
\frac{(\Gamma_i \vdash t_i : m \ \text{[Guard]})^i}{\Gamma_t, x : m_x \vdash t : \text{Return}}
\]

\[
\frac{\Gamma_u, x : m_x \vdash u : m}{\Gamma_u, x : m_x \vdash \text{let rec } x = t \ \text{in } u : m}
\]

\[
m_{x \in t} \leq \text{Guard}
\]

\[
m_{x \in u} \overset{\text{def}}{=} \max(m_{x \in u}, \text{Guard})
\]

\[
m'_{x \in u}[\Gamma_t] + \Gamma_u \vdash \text{let rec } x = t \ \text{in } u : m
\]
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.
Vicious $\overset{\text{def}}{=} \{ E_f[x] \mid \nexists v, (x = v)^{\text{ctx}} \in E_f \}$

**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.
Bonus slide: reduction example

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

\rightarrow
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)

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

\[
\rightarrow \text{match } \left( \begin{array}{l}
\text{let rec } xs = \text{Cons}(1, xs) \\
\text{Cons}(1, xs)
\end{array} \right) \text{ with } \left[ \\
\begin{array}{l}
\text{Nil } \rightarrow \text{None} \\
\text{Cons}(y, ys) \rightarrow \text{Some}(ys)
\end{array} \right]
\]
Bonus slide: reduction example

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

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

\[
\rightarrow
\]
Bonus slide: reduction example

(let rec \((x_i = v_i)^i\) in ...)
Bonus slide: reduction example

(let rec \(x_i = v_i\)^i \ in \ldots)

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

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

\[
\rightarrow \text{let rec } xs = \text{Cons}(1, xs) \text{ in }
\]
Bonus slide: reduction example

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

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

\[
\rightarrow \text{let rec } xs = \text{Cons}(1, xs) \text{ in }
\]
Bonus slide: reduction example

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

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

\[
\rightarrow \text{let rec } xs = \text{Cons}(1, xs) \text{ in } \text{Some}(xs)
\]
Bonus slide: Source term syntax

Terms \( \ni t, u ::= x, y, z \)
\[ \left| \begin{array}{l}
\text{let rec } b \text{ in } u \\
\lambda x. t \mid t u \\
K (t_i)^i \mid \text{match } t \text{ with } h
\end{array} \right. \]

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

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

ValueBindings \( \ni B ::= (x_i = v_i) \)

BindingCtx \( \ni L ::= \Box \mid \text{let rec } B \text{ in } L \)
Values $\ni \nu ::= \lambda x. t \mid K (w_i)^i \mid L[\nu]$  
WeakValues $\ni \omega ::= x \mid \nu \mid L[\omega]$  
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 $\ni F$

\begin{align*}
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{align*}
Values ⊆ v ::= λx. t | K (w_i)^i | L[v]
WeakValues ⊆ w ::= x | v | L[w]
ValueBindings ⊆ B ::= (x_i = v_i)^i
BindingCtx ⊆ L ::= □ | let rec B in L
EvalCtx ⊆ E ::= □ | E[F]
EvalFrame ⊆ F

\[
\begin{align*}
(x = v)^\text{ctx} & \in E \\
E[x] & \rightarrow E[v] \\
\hline
(t \rightarrow^\text{hd} t') & E[t] \rightarrow E[t']
\end{align*}
\]
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 \( \ni F \)

\[
\frac{(x = v) \in E}{E[x] \to E[v]} \quad \frac{t \to^{\text{hd}} t'}{E[t] \to E[t']} \quad \frac{(x = v) \in F \lor (x = v) \in E}{(x = v) \in E[F]} \quad \frac{(x = v) \in B}{(x = v) \in \text{let rec } B \text{ in } \square} \quad \frac{(x = v) \in (b \cup b')}{(x = v) \in \text{let rec } b, y = \square, b' \text{ in } u}
\]
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 \( \ni F \)

\[
\begin{align*}
(x = v) \in E & \quad \quad t \rightarrow^{\text{hd}} t' \\
E[x] & \rightarrow E[v] & E[t] & \rightarrow E[t']
\end{align*}
\]

\[
\begin{align*}
(x = v) \in B & \quad \quad (x = v) \in F \\
(x = v) \in \text{let rec } B \text{ in } \square & \quad \quad (x = v) \in (b \cup b') \\
(x = v) \in \text{let rec } b, y = \square, b' \text{ in } u
\end{align*}
\]

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

match \( L[K (w_i)^i] \) with \( (\ldots \mid K (x_i)^i \rightarrow u \mid \ldots) \rightarrow^{\text{hd}} L[u[(w_i/x_i)^i]] \)
ForcingFrame ⊨ F_f ::= □ v | v □
               | match □ with h
               | let rec b, x = □, b' in t

ForcingCtx ⊨ E_f ::= F_f | E[E_f] | E_f[L]

Vicious \ \overset{\text{def}}{=} \{ E_f[x] \mid \not\exists v, (x = v) \in E_f \}$
Bonus slide: mutual recursion

\[(x_i : \Gamma_i)^i \vdash \text{rec } b \quad (m'_i)^i \overset{\text{def}}{=} (\max(m_i, \text{Guard}))^i \quad \Gamma_u, (x_i : m_i)^i \vdash u : m\]

\[\sum (m'_i [\Gamma_i])^i + \Gamma_u \vdash \text{let rec } b \text{ in } u : m\]

\[(\Gamma_i, (x_j : m_{i,j})^j_{\in I} \vdash t_i : \text{Return})^i_{\in I} \quad (m_{i,j} \leq \text{Guard})^i_{,j}\]

\[\left(\Gamma'_i = \Gamma_i + \sum (m_{i,j} [\Gamma'_j])^j\right)^i\]

\[(x_i : \Gamma'_i)^i_{\in I} \vdash \text{rec } (x_i = t_i)^i_{\in I}\]