# MiniOCaml with lazy evaluation

- **In this lecture, we explore the impact of making one small change in the rules for evaluation via substitution:**

(Const) Const x $\Downarrow_\ell$ Const x

(Fun) Fun($a$,$e$) $\Downarrow_\ell$ Fun($a$,$e$)

(Rec) Rec($f$,Fun($a$,$e$)) $\Downarrow_\ell$ Fun($a$,$e$[Rec($f$,Fun($a$,$e$))/$f$]

($\delta$) $e\ op\ e' \Downarrow_\ell v\ OP\ v'$
$e \Downarrow_\ell v$
$e' \Downarrow_\ell v'$

($\delta$) $op\ e \Downarrow_\ell OP\ v$
$e \Downarrow_\ell v$

(If) If($e_1$, $e_2$, $e_3$) $\Downarrow_\ell v$
$e_1 \Downarrow_\ell$ True
$e_2 \Downarrow_\ell v$

(If) If($e_1$, $e_2$, $e_3$) $\Downarrow_\ell v$
$e_1 \Downarrow_\ell$ False
$e_3 \Downarrow_\ell v$

(List) $[e_1, \ldots, e_n] \Downarrow_\ell [v_1, \ldots, v_n]$
$e_1 \Downarrow_\ell v_1$
$\vdots$
$e_n \Downarrow_\ell v_n$

(App) $e\ e' \Downarrow_\ell v$
$e \Downarrow_\ell$ Fun($a$, $e''$)
$e''[e'/a] \Downarrow_\ell v$

# Lazy evaluation (cont.)

- Since all expressions being evaluated are closed, they have the same value regardless of *when* they are evaluated — whether before the function call or within the function call. For this reason, $\Downarrow$ and $\Downarrow_\ell$ almost always produce the same result. But there are exceptions:

```
(fun x -> 3) (4/0) ⇓
```

(App)

fun x → 3 ⇓ fun x → 3 (Fun)

4/0 ⇓ run-time error

```
(fun x -> 3) (4/0) ⇓ℓ 3
```

(App)

fun x → 3 ⇓ fun x ⇓ 3 (Fun)

3 ⇓ 3 (Const) (Note: $3[(4/0)/x] = 3$ )

# Lazy evaluation (cont.)

- **Another exception. This is the abstract syntax version of** "`let rec f x = f x in (fun x -> 3)(f 0)`":

```
let f = Rec(f, fun x -> f x) in (fun x->3)(f 0) ⇓
```

(See next page

Call this $\mathcal{F}$

```
let f = Rec(f, fun x -> f x) in (fun x->3)(f 0) ⇓ℓ
```

$\mathcal{F} \Downarrow \text{Fun}(x, \mathcal{F}\ x)$

$(\text{fun } x \to 3)((\text{Fun}(x, \mathcal{F}\ x))\ 0) \Downarrow 3$

$\text{fun } x \to 3 \Downarrow \text{fun } x \to 3$

$3 \Downarrow 3 \qquad (3[(\text{Fun}(x, \mathcal{F}\ x)\ 0)/x] = 3)$

Let $\mathcal{F}$ denote Rec(f, fun x → f x)

let f = $\mathcal{F}$ in (fun x → 3)(f 0) ⇓ ?

$\mathcal{F}$ ⇓ Fun(x, $\mathcal{F}$ x)

(fun x → 3) ((Fun(x, $\mathcal{F}$ x)) 0) ⇓ ?

fun x → 3 ⇓ fun x → 3

((Fun(x, $\mathcal{F}$ x)) 0) ⇓ ?

Fun(x, $\mathcal{F}$ x) ⇓ Fun(x, $\mathcal{F}$ x)

0 ⇓ 0

$\mathcal{F}$ 0 ⇓ ?

$\mathcal{F}$ ⇓ Fun(x, $\mathcal{F}$ x)

0 ⇓ 0

$\mathcal{F}$ 0 ⇓ ?

In a loop - will keep evaluating $\mathcal{F}$ 0, which has $\mathcal{F}$ 0 as a sub-calculation

# Removing features: lists (cont.)

Actually used $\Downarrow_l$ instead of $\Downarrow$, because the calculation is a little shorter.

[ `hd (cons 3 nil)` $\Downarrow$ 3

This is an abbreviation for:

(fun l → l (fun h t n → h))(fun f → f 3 (fun f → f 0 0 true) false) $\Downarrow$ 3

fun l → l (fun h t n → h) $\Downarrow$ fun l → l (fun h t n → h)

(fun f → f 3 (fun f → ...)(fun h t n → h) $\Downarrow$ 3

fun f → f 3 (fun f → ...) $\Downarrow$ fun f → f 3 (fun f → ...)

(fun h t n → h) 3 (fun f → f 0 0 true) false $\Downarrow$ 3

fun h t n → h $\Downarrow$ fun h t n → h

3 $\Downarrow$ 3

(Note: here we have used the obvious extension of our language to allow multiple arguments. E.g. for three arguments:

(Fun) fun x y z → e $\Downarrow$ fun x y z → e

(App) $e_0\ e_1\ e_2\ e_3 \Downarrow v$

$e_0 \Downarrow$ fun x y z → e

$e_i \Downarrow v_i$, $i = 1, 2, 3$

$e[v_1/x, v_2/y, v_3/z] \Downarrow v$ )