HW3 Part 1, List Operators

`Nil` represents the empty list.

Premises

$$\frac{\boxed{}}{\boxed{\text{Nil} \Downarrow \text{Nil}}}\ \textsc{Nil}$$

Conclusion

Evaluate Nil:

Nil -> Nil

Every time (no premises) you evaluate Nil, you will get another Nil!

HW3 Part 1, List Operators

`Cons(e1, e2)` represents the cons cell `e1 :: e2`, i.e., making a new list whose head is `e1` and whose tail is `e2`

Premises

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 :: e_2 \Downarrow v_1 :: v_2}\ \textsc{Cons}$$

Conclusion

Evaluate ListCons:

e1 -> v1

e2 -> v2

ListCons (e1, e2) -> ListCons(v1, v2)

It is **NOT**

ListCons (e1, e2) -> ListCons(e1, e2)!

(It is important in Part 1.4, and then...)

HW3 Part 1, List Operators

- `Match(e1, e2, e3)` represents list pattern-matching as in `match e1 with Nil -> e2 | x::xs -> e3 end` Note that the second branch of the match involves binding: the head of the list is bound to name "x" and the tail bound to "xs". Therefore, the third argument of the `Match` constructor will always be two nested `Scope` constructors.

2. Pattern-match on lists must end with the `end` keyword, unlike in OCaml.

Pattern Match

$$\frac{e_1 \Downarrow \mathsf{Nil} \qquad e_2 \Downarrow v}{\boxed{\mathsf{match}}\ e_1\ \boxed{\mathsf{with}}\ \mathsf{Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3\ \boxed{\mathsf{end}} \Downarrow v}\ \textsc{MatchNil}$$

$$\frac{e_1 \Downarrow v_1 :: v_2 \qquad e_3[x \mapsto v_1][y \mapsto v_2] \Downarrow v_3}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3\ \mathsf{end} \Downarrow v_3}\ \textsc{MatchCons}$$

HW3 Part 1, List Operators

- There are two inference rules, the whole semantic of `Match(e1, e2, e3)` is represented by those two rules:
- If e1 is Nil and e2 -> **`v`**, the evaluation result of the pattern match is **`v`**.
- If e1 is v1::v2 and e3[x->v1][y->v2] -> **`v3`**, the evaluation result of the pattern match is **`v3`**.

$$\frac{e_1 \Downarrow \mathsf{Nil} \qquad e_2 \Downarrow v}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \to e_2 \mid x :: y \to e_3\ \mathsf{end} \Downarrow v}\ \textsc{MatchNil}$$

$$\frac{e_1 \Downarrow v_1 :: v_2 \qquad e_3[x \mapsto v_1][y \mapsto v_2] \Downarrow v_3}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \to e_2 \mid x :: y \to e_3\ \mathsf{end} \Downarrow v_3}\ \textsc{MatchCons}$$

HW3 Part 1, What if ...?

What is call-by-value, and what is call-by-name?

```
    e1 ↓↓ \lambda x.e1'
    e2 ↓↓ v
    e1'[x -> v] ↓↓ v'
------------------------- (App)
    (e1 e2) ↓↓ v'
```

```
    e1 = \lambda x.e1'
    e2 ↓↓ v
    e1'[x -> v] ↓↓ v'
------------------------- (App-Alt1)
    (e1 e2) ↓↓ v'
```

Exhibit an expression e such that $\exists v.\ e \Downarrow v$ but $\neg\exists v.\ e \Downarrow_1 v$,

i.e., the evaluation of e works fine with the original rule but gets stuck/doesn't terminate with the alternative rule.

Consider e = `lambda x. lambda y. x + y`

HW3 Part 1, What if ...?

What is call-by-value, and what is call-by-name?

```
    e1 ↓↓ \lambda x.e1'
    e2 ↓↓ v
    e1'[x -> v] ↓↓ v'
------------------------ (App)
    (e1 e2) ↓↓ v'
```

```
    e1 ↓↓ \lambda x.e1'
    e1'[x -> e2] ↓↓ v'
-------------------------- (App-Alt2)
    (e1 e2) ↓↓ v'
```

Exhibit an expression e such that $\neg\exists v.\ e \Downarrow v$ but $\exists v.\ e \Downarrow_2 v$,

i.e., the evaluation of e gets stuck/doesn't terminate with the original rule but works fine with the alternative rule.

Consider e = `(lambda x. true) (1+true)`

HW3 Part 1, What if ...?

ListCons: Check the difference between those two inference rules.

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 :: e_2 \Downarrow v_1 :: v_2}\ \textsc{Cons} \qquad\qquad \frac{}{e_1 :: e_2 \Downarrow \boxed{e_1 :: e_2}}\ \textsc{Cons}$$

$$\frac{e_1 \Downarrow \mathsf{Nil} \qquad e_2 \Downarrow v}{\mathsf{match}\ e_1\ \mathsf{with\ Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3\ \mathsf{end} \Downarrow v}\ \textsc{MatchNil}$$

Incompatible!

$$\frac{\boxed{e_1 \Downarrow v_1 :: v_2} \qquad e_3[x \mapsto v_1][y \mapsto v_2] \Downarrow v_3}{\mathsf{match}\ e_1\ \mathsf{with\ Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3\ \mathsf{end} \Downarrow v_3}\ \textsc{MatchCons}$$

HW3 Part 1, What if ...?

```
------------------------ (Cons-Alt)
    e1::e2 ↓↓ e1::e2
```

```
   e1 ↓↓ h::t
   h ↓↓ vh
   t ↓↓ vt
     e3[x |-> h][y |-> t] ↓↓ v3
------------------------ (MatchCons-Alt)
   match e1 with
   | Nil -> e2
   | x::y -> e3
   end
   ↓↓ v3
```

Evaluate in difference rules

By compatible, we mean that if $e \Downarrow v$ using the original rules, then we should also have $e \Downarrow_1 v$ using the alternative rules. That is, for any expression e, the new rules should behave the same as the old rules if the old rules indeed evaluates e to some value v, although the new rules can be more permissive, i.e., e may get stuck with the old rules but works fine with the alternative rules.

As such, the human designer will always face a load of design decisions that may lead to systems with different theoretical properties and practical trade-offs.

HW3 Part 3, Semantics Reverse-Engineering

Remember that we have ***different*** design choices in the inference rules.

Have you ever tried to reverse-engineer the interpreter? `{ e1, e2 }`

```
{ 2, 4 } -> ?

{ 2, 2 + 2 } -> ?

{ 1 + 1, 2 + 2 } -> ?

{ (lambda x. x + 1) 3, 2 + 2 } -> ?

{ 2 + 2, (lambda x. x + 1) 3 } -> ?
```

Keeping one element changing, would you observe something?

HW3 Part 3, Semantics Reverse-Engineering

Remember that we have ***different*** design choices in the inference rules.

It’s the same for the `fst e`

```
fst { 2, 4 } -> ?

fst 2 -> ?

fst 2::2 -> ?

fst { 1 + 1, 2 + 2 } -> ?

fst { 1 + 1, (lambda x. x + 1) 3 } -> ?
```

Keeping one element changing, would you observe something?

HW3 Part 3, Semantics Reverse-Engineering

Remember that we have ***different*** design choices in the inference rules.

It's the same for the `snd e`

`snd { 2, 4 } -> ?`

`snd 2 -> ?`

`snd 2::2 -> ?`

`snd { 1 + 1, 2 + 2 } -> ?`

`snd fst { {1 + 1, 2 + 2}, 1 + 1 } -> ?`

Keeping one element changing, would you observe something?

HW3 Part 3, Semantics Reverse-Engineering

Write the inference rules backed by the information you collected in reverse-engineering.

The behavior of your inference rules shall remain the same as the interpreter's behavior!!!

HW3 Part 2, Augmenting the Interpreter, free_vars

IfThenElse:

- If (lambda x. y == 1) then (lambda x. z + 1) else (lambda x. w + 1)

ListMatch:

match (lambda x. y + 1)::(lambda x. z + 1) ->

Nil -> (lambda x. w + 1)

head::tail -> (head 1)::(tail 2)

HW3 Part 2, Augmenting the Interpreter, eval

```
let rec eval (e : expr) : expr =
 try
   match e with
    ...
    Fix (Scope (x, e')) -> ???
```

f -> ??

fix f -> ??

f |-> fix f -> ??

$$\frac{e[f \mapsto \text{fix } f \text{ is } e] \Downarrow v}{\text{fix } f \ \boxed{\text{is}}\ e \Downarrow v} \ \textsc{Fix}$$

Self-refer

HW3 Part 2, Augmenting the Interpreter, eval

```
let rec eval (e : expr) : expr =
 try
   match e with
     ...
     ListCons (e1, e2) -> ???
     ListMatch (e1, e2, Scope (x, Scope (y, e3))) -> ???
```

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 :: e_2 \Downarrow v_1 :: v_2}\ \textsc{Cons}$$

$$\frac{e_1 \Downarrow \mathsf{Nil} \qquad e_2 \Downarrow v}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3\ \mathsf{end} \Downarrow v}\ \textsc{MatchNil}$$

$$\frac{e_1 \Downarrow v_1 :: v_2 \qquad e_3[x \mapsto v_1][y \mapsto v_2] \Downarrow v_3}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3\ \mathsf{end} \Downarrow v_3}\ \textsc{MatchCons}$$