CS 421 Lecture 23 – Operational semantics

- Two versions of operational semantics, one without state and one with. (The one with state is for handling "ref" values.)
- OS_{clo}
- OS_{state}
- Scope rules
- how to handle recursion

Reminder: OS_{simp}

$$\frac{}{k \Downarrow k} \qquad \frac{}{\textbf{fun } x \text{ -> } e \Downarrow \textbf{fun } x \text{ -> } e}$$

$$\frac{e_1 \Downarrow \textbf{fun } x \text{ -> } e \qquad e_2 \Downarrow v' \qquad e[v'/x] \Downarrow v}{e_1 e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v = v_1 \oplus v_2}{e_1 \oplus e_2 \Downarrow v}$$

OS_{clo}

Use closures to represent function values – closer to actual implementation

Closure = abstraction * environment

Environment = variable $\rightarrow$ Value - (partial) functions from variables to values

In closure <e, η>, η contains values of free variables in e

Value = constants $\cup$ closures

Judgments:

$$\eta, e \Downarrow v$$

Note: unlike OS_{simp}, e can contain free variables – but they must be defined in η.

Examples of judgments

$$\varnothing, + \; 3 \; 4 \Downarrow 7$$
$$\{x \longmapsto 3\}, + \; x \; 4 \Downarrow 7$$
$$\{f \longmapsto \text{< fun a -> a+a, } \varnothing\text{>}\}, \; f \; 4 \Downarrow 8$$
$$\{f \longmapsto \text{< fun a -> a+b, } \{b \longmapsto 10\}\text{>}\}, \; f \; 4 \Downarrow 14$$

Note: unlike OS_{simp}, e can contain free variables – but they must be defined in η.

(Recursive functions will be discussed later.)

Rules of OS_{clo}

$$\text{Const}\ \frac{}{\eta, k \Downarrow k} \qquad\qquad \frac{}{\eta, x \Downarrow \eta x}\ \text{Var}$$

$$\frac{}{\eta,\ \text{fun } x \text{ -> } e \Downarrow\ \langle \text{fun } x \text{ -> } e, \eta \rangle}\ \text{Abs}$$

$$\frac{\eta,\ e_1 \Downarrow \langle \text{fun } x \text{ -> } e, \eta' \rangle \quad \eta, e_2 \Downarrow v' \quad \eta'[x \mapsto v'],\ e \Downarrow v}{\eta,\ e_1 e_2 \Downarrow v}$$

$$\frac{\eta,\ e_1 \Downarrow v_1 \quad \eta,\ e_2 \Downarrow v_2 \quad v = v_1 \oplus v_2}{\eta,\ e_1 \oplus e_2 \Downarrow v}$$

When evaluating $\eta, e_1 \Downarrow \langle \text{fun } x \to e, \eta' \rangle$,

η and η' may be different

```
let f = let x = 1
        in let y = 2
           in fun z -> x + y * z
```

fun z -> 1 + 2 * z

evaluated in $\{x \to 1, y \to 2\}$

```
in let x = 10
   in f 5
```

$\Rightarrow 11$

Evaluate in

$\{f \to \langle \cdots \rangle, x \to 10\}$

Example

$$\dfrac{\dfrac{}{\eta_1, \text{fun } y \to x+y \Downarrow \langle \text{fun } y \to x+y, \eta_1 \rangle}\,\text{Abs} \qquad \dfrac{}{\eta_1, 3 \Downarrow 3}\,\text{Const} \qquad \dfrac{\dfrac{}{\eta_2, x \Downarrow 4}\,\text{Var} \qquad \dfrac{}{\eta_2, y \Downarrow 3}\,\text{Var}}{\underbrace{\eta_1[y \to 3]}_{\eta_2}, x+y \Downarrow 7}}{\underbrace{\emptyset[x \to 4]}_{\eta_1}, (\text{fun } y \to x+y)\,3 \Downarrow 7}\,\text{App}$$

$$\dfrac{\underbrace{\emptyset}_{\eta}, \underbrace{\text{fun } x \to (\text{fun } y \to \ldots)}_{e_1} \Downarrow \langle \text{fun } \overbrace{x}^{x} \to \underbrace{\ldots}_{e}, \underbrace{\emptyset}_{\eta} \rangle \qquad \underbrace{\emptyset}_{\eta}, \underbrace{4}_{e_2} \Downarrow \underbrace{4}_{v'} \qquad \ldots}{\emptyset, \text{(fun x -> (fun y -> x+y) 3) 4} \Downarrow 7 \checkmark}\,\text{App}$$

OS_{state}

To handle side effects, we need to add "state". Unlike an environment, a state is something that changes during execution of the body of a function.

Expressions evaluate to a value, but also change the state.

Define a new set Loc = $\{\ell_0, \ell_1, \ell_2, \ldots\}$ of *locations*. A *state* σ is a map from Loc to Value.

Value = constants $\cup$ locations $\cup$ closures

Environment = variable $\to$ Value

Closure = abstraction * Environment

Judgments: $\sigma, \eta \vdash e \Downarrow v, \sigma'$

Rules of OS_{state}

Const

$$\frac{}{\sigma, \eta \vdash k \Downarrow k, \sigma} \qquad \frac{}{\sigma, \eta \vdash x \Downarrow \eta x, \sigma}$$

$$\frac{}{\sigma, \eta \vdash \textbf{fun } x \text{ -> } e \Downarrow \text{<}\textbf{fun } x \text{ -> } e, \eta\text{>}, \sigma}$$

$$\frac{\begin{array}{c}\sigma, \eta \vdash e_1 \Downarrow \text{<}\textbf{fun } x \text{ -> } e, \eta'\text{>}, \sigma_1 \\ \sigma_1, \eta \vdash e_2 \Downarrow v', \sigma_2 \\ \sigma_2, \eta'[x \mapsto v'] \vdash e \Downarrow v, \sigma'\end{array}}{\sigma, \eta \vdash e_1 e_2 \Downarrow v, \sigma'}$$

Rules of OS_{state}

$$\frac{\sigma,\eta \vdash e_1 \Downarrow v_1, \sigma_1 \quad \sigma_1,\eta \vdash e_2 \Downarrow v_2, \sigma' \quad v = v_1 \oplus v_2}{\sigma,\eta \vdash e_1 \oplus e_2 \Downarrow v, \sigma'}$$

Rules for stateful operators

$$\frac{\sigma,\eta \vdash e \Downarrow \ell, \sigma' \qquad \ell \in \mathrm{Loc} \quad \sigma'(\ell) = v}{\sigma,\eta \vdash\ !e \Downarrow v, \sigma'}$$

$$\frac{\sigma,\eta \vdash e_1 \Downarrow \ell, \sigma' \qquad \ell \in \mathrm{Loc} \qquad \sigma',\eta \vdash e_2 \Downarrow v, \sigma''}{\sigma,\eta \vdash e_1 := e_2 \Downarrow (), \sigma''[\ell \mapsto v]}$$

$()$

$$\frac{\sigma,\eta \vdash e \Downarrow v, \sigma' \qquad \ell \text{ a fresh location}}{\sigma,\eta \vdash \mathrm{ref}\, e \Downarrow \ell, \sigma'[\ell \mapsto v]}$$

Sequence optr: $;(v_1, v_2) = v_2$

Example

$$\{l \to 1\}, \eta_1[z \to ()] \vdash\ !x \Downarrow 1$$

$$\{l \to 0\}, \eta_1 \vdash \text{fun } z \to \ldots \Downarrow \langle \text{fun } z \to \ldots, \eta_1 \rangle \qquad \{l \to 0\}, \eta_1 \vdash y := !y+1 \Downarrow (), \{l \to 1\}$$

$$\{l \to 0\}, \underbrace{\{x \to l, y \to l\}}_{\eta_1} \vdash (\text{fun } z \to \ldots)(y := !y+1) \Downarrow 1, \{l \to 1\}$$

$$\{l \to 0\}, \{x \to l\} \vdash \text{fun } y \to \ldots \Downarrow \langle \text{fun } y \to \ldots, \{x \to l\} \rangle, \{l \to 0\} \qquad \{l \to 0\}, \{x \to l\} \vdash x \Downarrow l, \{l \to 0\}$$

$$\{\ell \mapsto 0\}, \{x \mapsto \ell\} \vdash (\text{fun } y \to \underline{(\text{fun } z \to !x)(y := !y+1)})x \Downarrow 1, \{\ell \mapsto 1\}$$

let x = ref 0 in

"Own" variables — like objects —
local state associated
with a function

```
let acc = let x = ref 0 in
        fun y -> (x := !x + y ; !x)

acc 1;;  => 1
acc 2;;  => 3
acc 2;;  => 5
   ⋮
```

Scope rules of OCaml

- Scope = which *definition* (let or fun or rec binding) is referred to at each *use* of a name
- Basic rule: whichever is lexically closest. This is a *static* rule – correspondence between definition and use is based solely on the program text.
- Note how the operational semantics enforces this.
- Java, C, C++ generally follow this rule for variables, with the exception of field references, which may refer to fields of a superclass; superclasses do not lexically enclose the reference. However, this rule is still static.

Dynamic scope

- Some languages use dynamic rules, where the correspondence between definition and use may vary during the course of execution.
- Example: dynamic binding of method calls in object-oriented languages
- LISP has higher-order ~~anonymous~~ functions, but uses dynamic scope, unlike OCaml.

Handling recursion

- Cannot create recursive functions with only abstraction and application – need to be able to use a name within its own definition

  let x = e in e' $\equiv$ (fun x -> e') e

(1) Could carry definitions around:

  Δ = map from function names to abstractions

  Judgments: Δ, e $\Downarrow$ v

(2) Better idea: introduce new function syntax:

  rec f e $\equiv$ function recursively defining f (e an abstraction)

  let rec f = e in e' $\equiv$ let f = rec f e in e'

  (then use let translation above)

Handling recursion (cont.)

- Advantage is that form of judgments, and existing rules, are retained.
- Evaluation rule for rec f e in OS_{simp}:

$$\frac{}{\mathbf{rec}\, f\ e \Downarrow e[\mathbf{rec}\, f\ e / f]}$$

Need rule for if

let rec f = fun x → if x=0 then 0 else f(x-1) in f 4

⇒ ~~let f = rec f (fun x → ...) in f 4~~

⇒ $\boxed{\text{(fun f → f 4)(rec f (fun x ...))}}$

→ (rec f (fun x→...)) 4

→ (fun x → if x=0 then 0 else (rec f (fun x →..))(x-1)) 4

→ if 4=0 then 0 else ...

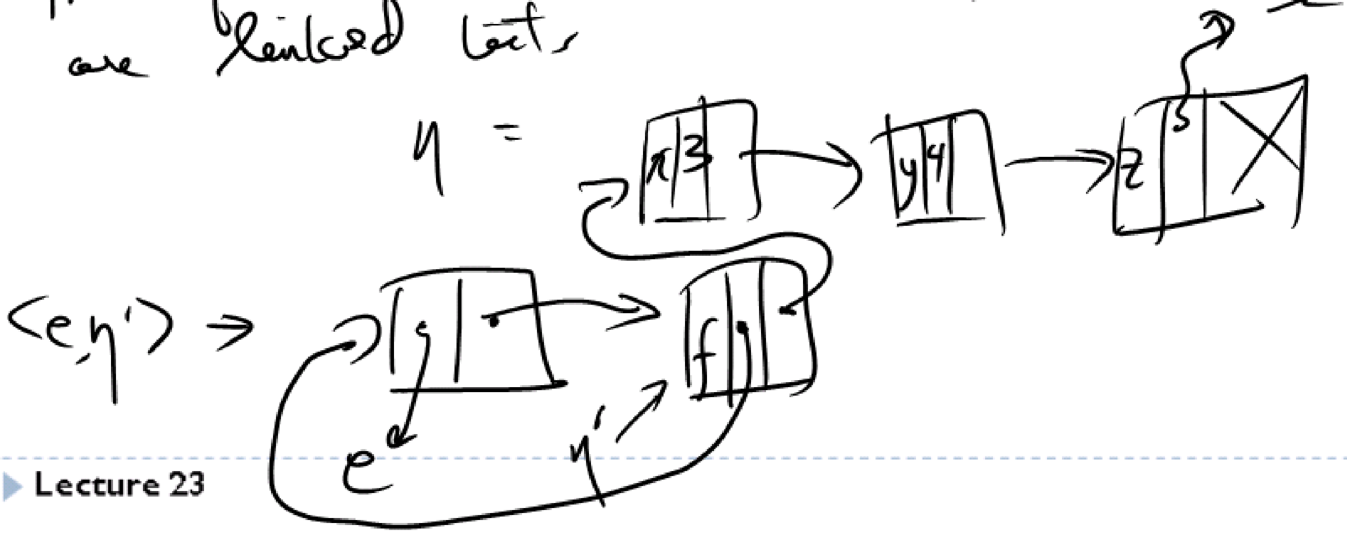
→ (ref f (fun x →..)) (4-1)

Handling recursion in OS$_{clo}$

$$\frac{}{\eta, \mathbf{rec}\, f\ e \Downarrow \langle e, \eta' \rangle}\ \text{(Rec)}$$

where η' is defined circularly such that $\eta' = \eta[f \mapsto \langle e, \eta' \rangle]$

Think of closures and env's as objects - env's are linked lists



Recursion in OS_{state}

$$\overline{\sigma,\eta \vdash \mathbf{rec}\, f\ e \Downarrow <e,\eta'>,\sigma}$$

where η' is defined circularly such that $\eta' = \eta[f \mapsto <e, \eta'>]$

Why formalize type system and operational semantics

- Unambiguous definition for language users and implementers
- Prove that definitions make sense

  E.g. $\boxed{\text{prove}}$ that type system and op. sem. are mutually consistent

Soundness: $\emptyset \vdash e : \tau$ & $e \Downarrow v$

$\Rightarrow \underbrace{v \text{ has type } \tau}_{\text{for } OS_{\text{simp}},\ v : \tau}$