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.)
- First, how to handle recursion
- OS_{clo}
- OS_{state}
- Scope rules

Reminder: OS_{simp}

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

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

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

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}:

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

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

$$\emptyset, + \ 3 \ 4 \Downarrow 7$$

$$\{x \mapsto 3\}, + \ x \ 4 \Downarrow 7$$

$$\{f \mapsto \text{< fun a -> a+a, } \emptyset\text{>}\}, \text{ f } 4 \Downarrow 8$$

$$\{f \mapsto \text{< fun a -> a+b, } \{b \mapsto 10\}\text{>}\}, \text{ f } 4 \Downarrow 14$$

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}

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

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

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

$$\frac{\eta,\ e_1 \Downarrow v_1 \qquad \eta,\ \mathrm{e}_2 \Downarrow v_2 \qquad v = v_1 \oplus v_2}{\eta,\ e_1 \oplus e_2 \Downarrow v}$$

Example

$$\frac{}{\varnothing, \text{(fun x -> (fun y -> x+y) 3) 4} \Downarrow 7}$$

Handling recursion in OS_{clo}

$$\frac{}{\eta, \text{rec}\, f\ \ e \Downarrow\ <e, \eta'>}\ \text{(Rec)}$$

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

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 $\rightarrow$ Value

Closure = abstraction * Environment

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

Rules of OS$_{\text{state}}$

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

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

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

Rules of OS_{state}

$$\frac{\sigma,\eta \vdash e_1 \Downarrow v_1, \sigma_1 \qquad \sigma_1,\eta \vdash \mathrm{e}_2 \Downarrow v_2, \sigma' \qquad v = v_1 \oplus v_2}{\sigma,\eta \vdash e_1 \oplus e_2 \Downarrow v, \sigma'}$$

$$\frac{}{\sigma,\eta \vdash \mathrm{rec}\, f\ e \Downarrow \ <e,\eta'>, \sigma}$$

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

Rules for stateful operators

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

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

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

Example

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

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