CS51 **Spring 2024**

Code Review 9: Semantics

1 Purpose of Semantics

In this class, we've mostly been focused on unit testing to **verify** the behavior of programs. **Formal Verification proves** the correctness of a program using **mathematics**.

Semantics helps us formally verify programs by describing **mathematically what programs mean**.

Using **mathematical rigor** allows us to because they allow us to provide a **precise** and **succinct** definition of how a programming language works. For example, some language specifications are described in books spanning hundreds of papers. With mathematical formalism, languages can be described in far fewer pages with the same level (or even more) precision. **Read the start of chapter 13 for a discussion of why semantics are helpful**.

In this course, we will look at **large-step**, **operational** semantics. **Operational** means we are concerned with what expressions **evaluate** to. **Large-step** semantics directly characterizes the **relation** between expressions and the values they evaluate to (**small-step** semantics describe what happens to an expression after each small step of evaluation).

Remark 1.1 (Notation)**.** With semantics, many students, at first, initially get overwhelmed by notation, especially if they haven't had much experience with mathematical formalism.

If you find yourself being overwhelmed, remember that we are simply using mathematics and notations to precisely define the evaluation of programs. We suggest that you make sure you understand what intuitively different rules mean, and then proceed to look at how the notation expresses that meaning.

In this course, we'll focus on a **large-scale operational semantics** for `OCaml`-like expressions. We'll look at two models: **substitution** semantics and **environmental** semantics.

Notation 1.2 (Judgment)**.** An evaluation **judgment** is an assertion of what expression evaluate to. Notationally, we say $P \Downarrow v$ if an expression P evaluates to value v.

2 Substitution Semantics

Substitution Semantics models what expressions evaluate to based on the substitution of expressions for **free variables**.

2.1 Free Variables

Definition 2.1 (Free Variable)**.** A **free variable** is a **variable** not bound by a **binding construct**. In an `OCaml` context, free variables are variables not bound by a `let` or a `fun` construct. **See section 13.3.2** in the textbook.

Example 2.2 (Free Variables)**.** Consider the following expression:

```
fun x -> let x = y in x + 3
```

`y` is the only free variables in this expression. We see that `fun` binds `x` and `let` binds `x`. The `x` in `x + 3` refers to the `x` bounds by `let`. We see that `y` is not bound by any binding construct, so it is free.

Notation 2.3 (Free Variable Rules)**.** The textbook formally defines rules for finding free variables.

$$FV(\overline{m}) = \varnothing \quad \text{(integers)} \tag{13.1}$$
$$FV(x) = \{x\} \quad \text{(variables)} \tag{13.2}$$
$$FV(P \texttt{ + } Q) = FV(P) \cup FV(Q) \quad \text{(and similarly for other binary operators)} \tag{13.3}$$
$$FV(P\ Q) = FV(P) \cup FV(Q) \quad \text{(applications)} \tag{13.4}$$
$$FV(\texttt{fun } x \texttt{ -> } P) = FV(P) - \{x\} \quad \text{(functions)} \tag{13.5}$$
$$FV(\texttt{let } x \texttt{ = } P \texttt{ in } Q) = (FV(Q) - \{x\}) \cup FV(P) \quad \text{(binding)} \tag{13.6}$$

Figure 13.3: Definition of FV, the set of free variables in expressions for a functional language with naming and arithmetic.

Figure 1: Free Variable Rules (see section B.5 of the textbook for set notation)

The rules follow intuitively from the definition of free variables. Integer expressions have no free variables. For an expression with one variable, the free variable is just the variable. `FV(let x = P in Q)` is just equal to the union of the free variables in `P` and the free variables in `Q` excluding `x` since `x` is bound by the `let` construct.

When finding free variables **by hand**, it's probably easier to rely on the intuitive definition; however, these rules are helpful for writing code to find free variables programmatically, which you'll do in the final project.

2.2 Substitution

Notation 2.4 (Substitution)**.** We describe notation that describes how to substitute expressions for free variables.

$$\overline{m}[x \mapsto Q] = \overline{m}$$
$$x[x \mapsto Q] = Q$$
$$y[x \mapsto Q] = y \quad \text{where } x \neq y$$
$$(P \texttt{ + } R)[x \mapsto Q] = P[x \mapsto Q] \texttt{ + } R[x \mapsto Q]$$
$$\text{and similarly for other binary operators}$$
$$(\texttt{let } y \texttt{ = } D \texttt{ in } B)[x \mapsto Q] = \texttt{let } y \texttt{ = } D[x \mapsto Q] \texttt{ in } B[x \mapsto Q]$$

Figure 2: Substitution Rules

The rules in figure 2 describe substitution for different types of expressions (**full rules on page 217, Figure 13.4 of the textbook**).

For example, $x[x \mapsto Q] = Q$ symbolizes that we substitute Q in for `x` for the free occurrences of `x` in the expression `x`.

Example 2.5 (Substitution Example)**.** Consider the following substitution:

$$(\text{let } x = y * y \text{ in } x + x)\,[y \mapsto 3] = \text{let } x = 3 * 3 \text{ in } x + x$$

Here, we substitute the free 3 in for the free occurrences of y, and we have two free occurrences of y.

2.3 Evaluation Rules

Now that we have described substitution, we can write down rules for evaluating expressions.

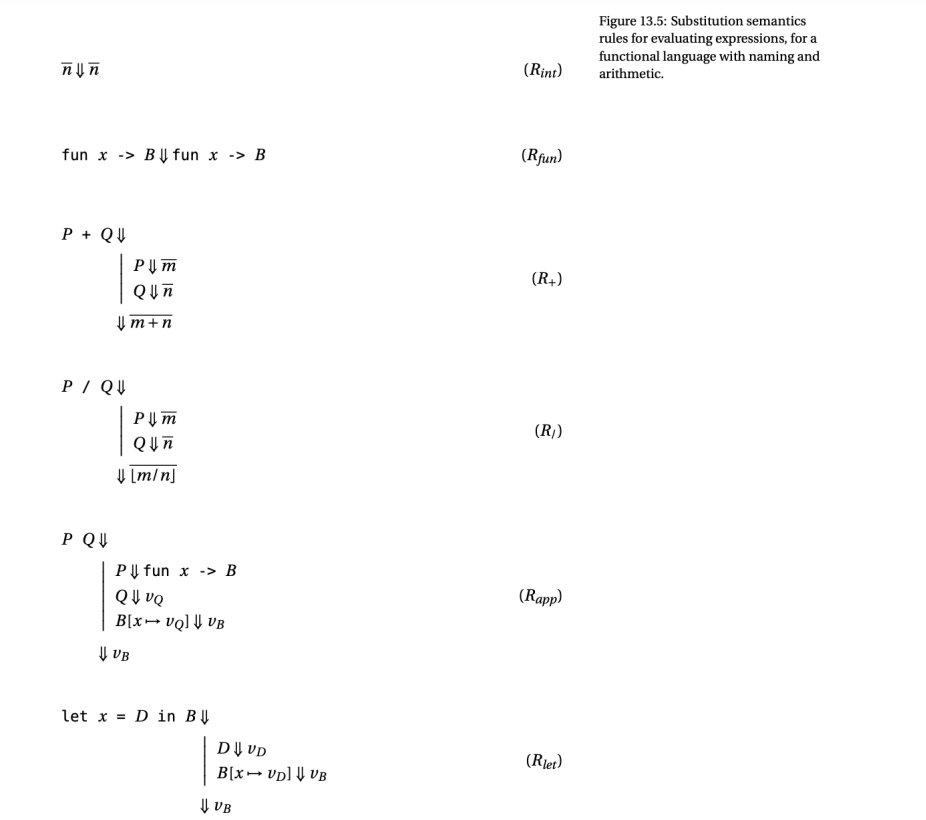

Figure 13.5: Substitution semantics rules for evaluating expressions, for a functional language with naming and arithmetic.

Figure 3: Evaluation Rules under Substitution Semantics (**See Figure 13.5 on page 220 for full rules**)

The R_{let} rule, for example, says that if D evaluates to v_D and the substitution $B[x \mapsto v_d] \Downarrow v_B$, then $letx = D$ in B evaluates to v_B.

3 Environment Semantics

Definition 3.1 (Environment)**.** An **environment** is a **mapping** from **variables** to **values**.

For example, the environment $\{x \mapsto 5; f \mapsto$ `fun x -> x`$; y \mapsto \ell\}$ signifies that x, y and z map to the values 5, `fun x -> x`, and ℓ, respectively (note ℓ could represent a location, as references are values and evaluate to a location).

In the environment semantic model, instead of substituting values for variables, we use environments.

Notation 3.2 (Environments)**. See Section 19.2 for full notes on how notation is defined**. $E(x)$ is the value that environment E maps an environment x to.

3.1 Lexical and Dynamic Environment Semantics

Definition 3.3 (Lexical Environment)**.** The environment in force when a function is **defined**.

Definition 3.4 (Dynamic Environment)**.** The environment in force when a function is **applied**.

Example 3.5 (Lexical vs Dynamic Semantics Example)**.** Consider the following expression:

```
let x = 2 in
let f = fun y -> x + y in
let x = 8 in
f x ;;
```

The dynamic environment of function f maps x (in the function body) to 8 and the lexical environment maps x to 2.

Dynamic Environment Semantics use the dynamic environment when evaluation function applies, while lexical environment semantics use the lexical environment.

This expression evaluates to 10 under lexical semantics and 16 under dynamic semantics.

Figure 19.1: Dynamic environment semantics rules for evaluating expressions, for a functional language with naming and arithmetic.

$$E \vdash \overline{n} \Downarrow \overline{n} \quad (R_{int})$$

$$E \vdash x \Downarrow E(x) \quad (R_{var})$$

$$E \vdash \texttt{fun } x \texttt{ -> } P \Downarrow \texttt{fun } x \texttt{ -> } P \quad (R_{fun})$$

$$\frac{E \vdash P \Downarrow \overline{m} \quad E \vdash Q \Downarrow \overline{n}}{E \vdash P \texttt{ + } Q \Downarrow \overline{m+n}} \quad (R_{+})$$

(and similarly for other binary operators)

$$\frac{E \vdash D \Downarrow v_D \quad E\{x \mapsto v_D\} \vdash B \Downarrow v_B}{E \vdash \texttt{let } x \texttt{ = } D \texttt{ in } B \Downarrow v_B} \quad (R_{let})$$

$$\frac{E \vdash P \Downarrow \texttt{fun } x \texttt{ -> } B \quad E \vdash Q \Downarrow v_Q \quad E\{x \mapsto v_Q\} \vdash B \Downarrow v_B}{E \vdash P \ Q \Downarrow v_B} \quad (R_{app})$$

Figure 4: Dynamic Environment Rules

Figure 19.4: Lexical environment semantics rules for evaluating expressions, for a functional language with naming, arithmetic, and mutable storage.

$$E, S \vdash \overline{n} \Downarrow \overline{n}, S \quad (R_{int})$$

$$E, S \vdash x \Downarrow E(x), S \quad (R_{var})$$

$$E, S \vdash \texttt{fun } x \texttt{ -> } P \Downarrow [E \vdash \texttt{fun } x \texttt{ -> } P], S \quad (R_{fun})$$

$$\frac{E, S \vdash P \Downarrow \overline{m}, S' \quad E, S' \vdash Q \Downarrow \overline{n}, S''}{E, S \vdash P \texttt{ + } Q \Downarrow \overline{m+n}, S''} \quad (R_{+})$$

(and similarly for other binary operators)

$$\frac{E, S \vdash D \Downarrow v_D, S' \quad E\{x \mapsto v_D\}, S' \vdash B \Downarrow v_B, S''}{E, S \vdash \texttt{let } x \texttt{ = } D \texttt{ in } B \Downarrow v_B, S''} \quad (R_{let})$$

$$\frac{E_d, S \vdash P \Downarrow [E_l \vdash \texttt{fun } x \texttt{ -> } B], S' \quad E_d, S' \vdash Q \Downarrow v_Q, S'' \quad E_l\{x \mapsto v_Q\}, S'' \vdash B \Downarrow v_B, S'''}{E_d, S \vdash P \ Q \Downarrow v_B, S'''} \quad (R_{app})$$

Figure 5: Lexical Environment Rules

Definition 3.6 (Closure)**.** Consider the R_{fun} rule for functions under lexical semantics: $E, S \vdash$ `fun x -> P` $\Downarrow [E \vdash$ `fun x -> P`$], S$.

So that the function is evaluated under its **lexical scope**, functions evaluate to **closures**, objects that store the function itself and the lexical environment of the function.

Example 3.7 (Recursion in Lexically-Scoped Environment Semantics)**.**

$E \vdash$ `let rec x =` D `in` B

1. Let environment E' extend E
   with a mutable binding of `x` to `Unassigned`
2. Evaluate D in E' to get a value v_D
   (possibly capturing E' in closures)
3. Change the value stored for `x` in E' to v_D
4. Evaluate B in the modified E'

Figure 6: Let Rec for Lexically-Scoped Environment Semantics

3.2 Mutable Storage

We can extend environmental semantics to handle imperative programming.

Example 3.8 (Store)**.** A **store** is a mapping from **locations** to **values**.

References evaluate to locations, which are `values`. A store would map those locations to other values (which may also be other locations).

Figure 19.4: (continued) Lexical environment semantics rules for evaluating expressions, for a functional language with naming, arithmetic, and mutable storage.

$$E, S \vdash \texttt{ref}\ P \Downarrow$$
$$\Big|\ E, S \vdash P \Downarrow v_P, S' \qquad (R_{ref})$$
$$\Downarrow l, S'\{l \mapsto v_P\} \qquad \text{(where } l \text{ is a new location)}$$

$$E, S \vdash\ !\ P \Downarrow$$
$$\Big|\ E, S \vdash P \Downarrow l, S' \qquad (R_{deref})$$
$$\Downarrow S'(l), S'$$

$$E, S \vdash P\ \texttt{:=}\ Q \Downarrow$$
$$\Big|\ E, S \vdash P \Downarrow l, S'$$
$$\Big|\ E, S' \vdash Q \Downarrow v_Q, S'' \qquad (R_{assign})$$
$$\Downarrow (), S''\{l \mapsto v_Q\}$$

$$E, S \vdash P\ \texttt{;}\ Q \Downarrow$$
$$\Big|\ E, S \vdash P \Downarrow (), S'$$
$$\Big|\ E, S' \vdash Q \Downarrow v_Q, S'' \qquad (R_{seq})$$
$$\Downarrow v_Q, S''$$

Figure 7: Lexical Semantic Rules for Mutable Storage

4 Practice Problems

Problem 4.1 (Free Variables)**. Instructions**: Write a box around the free variables in the following expressions. For the bound variables, draw an arrow indicating its corresponding binding occurrence.

1.
```
let x = y + 2 in
let x = x + 3 in
in let f = (fun z -> y + 5)
in let z = 8 in f z
```

2.
```
fun z -> a + b + c + z
```

3.
```
x (fun x -> fun x -> x (fun x -> x))
```

4.
```
 let y = y + 2 in
 let g = (fun x -> y + 5)
 in (fun z -> z d) g
```

5.
```
 let x = 5 in
 x + (fun a -> z (fun z -> a y)) 5
```

Problem 4.2 (Substitution)**.** Write out the result of the following substitutions.

1.
```
((x + 1) + (y + 2)) [x -> 3]
```

2.
```
(let x = x + 1 in x + 2) [x -> 3]
```

3.
```
(let x = y + 2
in let z = 5
in z + x + j a)[x -> 5][j -> (fun x -> x * x)][a -> 4]
```

4.
```
 (let y = y + 2 in
 let g = (fun x -> y + 5)
 in (fun z -> z d) g))[y -> 8][d -> 4]
```

5.
```
 (let x = 5 in
 x + (fun a -> z (fun z -> a y)) 5)[a -> 2][z -> fun x -> x]
```

Problem 4.3 (Substitution Semantics)**.** Use the Substitution Semantic Evaluation Rules to derive the result of the following expression.

```
let x = 8 in
let y = x * 3 in
(fun x -> x + 4) 5 + x + y
```

Problem 4.4 (Environment Semantics)**.**

1. Define an expression that evaluates to different values under lexical and dynamic semantic systems
2. Write out two derivations for the result of the expression: one using lexical semantics and the other using dynamic semantics.
3. Derive the evaluation for the result of the following expression

```
let f = (fun x -> x * 2) in
let x = ref 42 in
(x := !x - 10; !x) + f !x ;;
```

Problem 4.5 (Extending Semantics Models)**.** Extend the substitution semantics and environment semantics (under mutable storage) models to support the additional language constructs:

1. Tuple Pairs. In addition, define semantic rules for the functions `fst` and `snd`.
2. Conditional Expressions. Define semantic rules for operators `||` and `&&`. In addition, define a semantic rule for the `if e1 then e2 else e3` construct
3. Lists. Define semantic rules for the `hd` and `tl` functions, and the `::` constructor.