Chapter 1

Operational Semantics

I am not a fully operational person

Cliff

1.1 Introduction

Now that we can design a language, we may want to do a few of things, two of which we will talking about here:

- Give meaning to the language
- Prove correctness of a program

Both of these goals can be achieved through the use of operational semantics. Semantics referring to the meaning of a statement, and operational referring to how something operates.

1.2 Meaning

If you ever take a philosophical linguistics course[1] you talk about some weird things that happen in languages, but you also talk about how meaning is sometimes attached to words. Slang in particular falls in and out of favor so figuring out how we attach additional meaning to words is always brought up. How would you define "vibe" to a non-native speaker when saying something like "Did not pass the vibe check"? How would you describe "mid" in a sentence like "Cliff was pretty mid last semester"? Operational semantics is a way to help describe the meaning of a statement in a programming language. Analogously, how do you describe the sentence `'fun x -> x 3'` to someone unfamiliar with functional programming?

[1] Would recommend Phil360: Philosophy of Language with Alexander Williams

There's plenty of ways that you can describe meaning. In programming language theory there are typically three major ways: denotations semantics, axiomatic semantics and operational semantics.

- Denotations: describe meaning via mathematical constructs
- Operational: describe meaning via how something operates
- Axiomatic: describing meanings via axioms

How I think about (and I am sure that people more in both the linguistics and PL space would be mad at me) is that denotational semantics is by giving a definition. For example: '"Blue" refers to light waves that fall in-between 450 and 495 nm'. Axiomatic semantics gives examples. For example: 'the sky, the ocean, and that person's eyes are blue'. Operational semantics describe how we use it. Example: '"Blue" is referring to a shade people see between green and violet'.

So when we talk about the meaning of a program, we want to talk about it in terms of how the program operates. More specifically, we use operational semantics to communicate language design ideas. If we want to talk about another language however let's use some terms to help us. If I want to talk about some language x, then I will refer to x as the target language. The language that I will be describing x in, I will call the Meta-language. So If I want to talk about OCaml, then OCaml will be the target language, and English will be the Meta-language.

1.3 Correctness

When we talk about correctness, we basically mean, does the program run how we expect it to run? Can I prove that `+ 2 3` returns 5 in `Math-ew`? How can I prove that `(+ 2 (* 3 4))` returns 14 in LISP? The answer to this is not much different than proving that $((p \wedge q) \wedge (p \implies r) \wedge (q \implies r)) \rightarrow r$. Namely, we can make a proof:

$$\begin{array}{ll} & p \wedge q \\ & p \implies r \\ & q \implies r \\ \hline \therefore & r \end{array}$$

That is, if we know rules of things, we can derive new things. Suppose that we know that $3 * 4 = 12$ and we know that $12 + 2 = 14$. If we know these rules that we can say something like $2 + 3 * 4 = 14$ or `(+ 2 ( * 3 4))` returns 14. However instead of using defined rules of algebra or logic that we know, we are going to use defined rules of the target language.

1.4 Operational Semantics

Let us define a very basic language $Alanguge$:

$$\begin{array}{lll} e & \rightarrow & n \\ & \rightarrow & e\ ?\ e \\ n & \rightarrow & 0|1|2|3|... \end{array}$$

A has really two statements that exist in the language. Let us make a rule that describes what we should do when met with either of these two statements.

If the statement in A is just n, then I want to evaluate to myself. So 3 should evaluate to 3, and 5 should evaluate to 5. This rule is pretty basic and so we could say this is an axiom in our language, or that we don't need proof to say that 3 is 3. So we use the following notation:

$$\frac{}{n \Rightarrow n}$$

This is just a conclusion, or something that is true in and of itself.

On the other hand, if I am given a statement that looks like 3?4 I want something to be evaluated to a value. In A, I want to use ? to add its two operands. So I may need to have a rule that describes my two operands, and what I should do when I see something that looks like e ? e.

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2 \qquad n_3 \text{ is } n_1 + n_2}{e_1 \; ? \; e_2 \Rightarrow n_3}$$

This is to say that e_1 and e_2 are some expressions, and that e_1 will eventually evaluate to some number, n_1, while e_2 will eventually evaluate to some number n_2. We then need to describe that we want to add the two numbers together to get a final value: n_3 is $n_1 + n_2$. This part is described in our meta language. We then want to show that if these statements are true, then when we see e_1 ? e_2 that we want to return n_3, whatever that is.

This particular example that uses ? instead of +, is just to show you that we can just arbitrarily use symbols to stand for symbols and as long as we describe what this symbol does in our target language, then we can show you a rule of what is supposed to happen.

Now that we have our two rules to match each thing in our grammar, we can start making proofs that show what would happen if we had a statement like 4 ? 3.

In this example, looking at our grammar, we can see that 4 is e_1 and that 3 is e_2. We also know that, 4 and 3 are just numbers which we know evaluate to themselves. So constructing a proof of correctness for the statement 4 ? 4 using the above two rules would look like:

$$\frac{\dfrac{}{4 \Rightarrow 4} \qquad \dfrac{}{3 \Rightarrow 3} \qquad 7 \text{ is } 4 + 3}{4 \; ? \; 3 \Rightarrow 7}$$

That also means that we could prove larger expressions such as 3?4?5. Here I will assign 3 to e_1 and 4?5 to e_2, but you could instead say that 3?4 is e_1 and 5 is e_2. For this particular rule it does not matter, but depending on the operation, you may need to give more information so you don't get this ambiguous parse. The proof is as follows:

$$\frac{\dfrac{}{3 \Rightarrow 3} \qquad \dfrac{\dfrac{}{4 \Rightarrow 4} \qquad \dfrac{}{5 \Rightarrow 5} \qquad 9 \text{ is } 4+5}{4 \; ? \; 5 \Rightarrow 9} \qquad 12 \text{ is } 3 + 9}{3 \; ? \; 4 \; ? \; 5 \Rightarrow 12}$$

As we add more to our language, we need to add more rules to our operational semantics. Let us consider the language $BIanguage$:

$$\begin{aligned} e &\rightarrow n \\ &\rightarrow e + e \\ &\rightarrow V \\ &\rightarrow \text{let } V = e \text{ in } e \\ n &\rightarrow 0|1|2|3|\ldots \\ V &\Rightarrow a|b|c|d|\ldots \end{aligned}$$

I have changed the ? symbol to a + since we know that we just want to add the subexpressions anyway. Additionally, we have now added variables to our language. By adding variables, we need to add something to our operational semantics: an environment.

Simply put, an environment is a mapping from variables to values.An example environment could be something like `[x:3,y:4]` We will denote an arbitrary environment with the character A. We will also need to update our rules to incorporate this environment. Let's first update our rules. The updated number and + rule are:

$$\frac{}{A; n \Rightarrow n}$$

$$\frac{A; e_1 \Rightarrow n_1 \qquad A; e_2 \Rightarrow n_2 \qquad n_3 \text{ is } n_1 + n_2}{A; e_1 \ ? \ e_2 \Rightarrow n_3}$$

What this means is that each expression e_x is being evaluated with the environment A. So suppose that we have previously bound the variable x is to the value 4. If we want to evaluate the statement 6 with this environment, then the proof would look like:

$$\frac{}{A, x : 4; 6 \Rightarrow 6}$$

I still include A because there are probably other environment variables that we are unaware of.

However, this is quite a boring example. What we may care about is how to look up a variable in our language. That is, what is the rule for $e \longrightarrow V$? If we want to evaluate V into a value, we need to look up that value in the environment. Thus, our rule has to describe this process. Conventionally, we do this in the following way:

$$\frac{A(x) \Rightarrow v}{A; x \Rightarrow v}$$

So if had previously bound the variable x to the value 4 and wanted to look up x, it would look like:

$$\frac{A, x : 4; (x) \Rightarrow 4}{A, x : 4; x \Rightarrow 4}$$

This rule looks like it's just a repetition of a line, but recall the idea of the target language and the meta language. The conclusion is describing the target language, while the premise or hypothesis is describing what to do in the meta language.

We did do this a bit out of order. Before we can look up anything, we would have first needed to bind something. So let's describe the rule of $e \longrightarrow$ let $V = e_1$ in e_2.

In this case, following OCaml (this is not always the case), before we bind a value to a variable, we want to evaluate the expression e_1 to a value and then bind that resulting value to the variable. Then we want to use this new binding when we are evaluating the expression e_2. Consider `let x = 3 in x + 1`. x+1 is the body and the binding we just made `x = 3` should be used when evaluating this. The rule that describes all this is the following:

$$\frac{A; e_1 \Rightarrow v \qquad A, x : v; e_2 \Rightarrow e_3}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow e_3}$$

So in this case, we are evaluating e_1 to a value v, and then adding this binding to the environment when we evaluate e_2.

Using these rules, let us show a proof of correctness that let $x = 3$ in $x + 4$.

$$\frac{A; e_1 \Rightarrow v \qquad A, x : v; e_2 \Rightarrow e_3}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

In this case we identify that 3 is e_1 and $x + 4$ is e_2.

$$\frac{A; 3 \Rightarrow v \qquad A, x : v; x + 4 \Rightarrow e_3}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

We know that $\frac{}{3 \Rightarrow 3}$ so

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad A, x : 3; x + 4 \Rightarrow e_3}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

We then want to use our plus rule when evaluating $x + 4$

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad \frac{A,x:3;e_4 \rightarrow n_1 \qquad A,x:3;e_5 \rightarrow n_2 \qquad n_3 \text{ is } n_1 + n_2}{A,x:3;x+4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

Here we can do what we did above an notice that in $x + 4$ that x is e_4 and 4 is e_5.

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad \frac{A,x:3;x \rightarrow n_1 \qquad A,x:3;4 \rightarrow n_2 \qquad n_3 \text{ is } n_1 + n_2}{A,x:3;x+4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

Based on our variable lookup rule we can say that $x \Rightarrow 3$ making $n_1 = 3$:

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad \frac{\frac{A,x:3;(x) \Rightarrow 3}{A,x:3;x \rightarrow 3} \qquad A,x:3;4 \rightarrow n_2 \qquad n_3 \text{ is } 3 + n_2}{A,x:3;x+4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

We know that 4 evaluates to itself:

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad \frac{\frac{A,x:3;(x) \Rightarrow 3}{A,x:3;x \rightarrow 3} \qquad \frac{}{A,x:3;4 \rightarrow 4} \qquad n_3 \text{ is } 3 + 4}{A,x:3;x+4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

and we know that $3 + 4$ is the value of 7:

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad \frac{\frac{A,x:3;(x) \Rightarrow 3}{A,x:3;x \rightarrow 3} \qquad \frac{}{A,x:3;4 \rightarrow 4} \qquad 7 \text{ is } 3 + 4}{A,x:3;x+4 \Rightarrow e_3}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow e_3}$$

Thus we know that e_3 is the final value of the expression.

$$\frac{\frac{}{A;3 \Rightarrow 3} \qquad \frac{\frac{A,x:3;(x) \Rightarrow 3}{A,x:3;x \rightarrow 3} \qquad \frac{}{A,x:3;4 \rightarrow 4} \qquad 7 \text{ is } 3 + 4}{A,x:3;x+4 \Rightarrow 7}}{\text{let } x = 3 \text{ in } x + 4 \Rightarrow 7}$$

One point of confusion is what happens when we have a statement like `let x = 3 in let x = 4 in x + 5`. We would eventually get to a point where we have to evaluate $A, x : 3, x : 4; x$. The question of which x you use is dependent on the the rules you have. In this case, I am adding any new binding to the end of A ($A, x : v; e$) which means in order to get the scope correct, I need to choose the right most binding in the list.

Now that we have some more rules, we can keep adding things to to our grammar (to our language) and write more rules for how they should act.

Let us consider the language $Clanguage$ (not to be confused with C):

$$\begin{aligned} e &\rightarrow n \\ &\rightarrow e + e \\ &\rightarrow V \\ &\rightarrow \text{let } V = e \text{ in } e \\ &\rightarrow B \rightarrow \\ if\,ethenelseen &\rightarrow 0|1|2|3|\ldots \\ V &\Rightarrow a|b|c|d|\ldots B \Rightarrow true|false \end{aligned}$$

We should add some rules to incorporate these new values.

TODO, but enough to post notes

1.5 Definitions interpreter

Let us go back to our very simple $Alanguage$. Recall the idea of Operational Semantics is to give meaning through how expressions operate. This is the basis of the idea of an interpreter. How a statement should evaluate is what the interpreter does. Thus, we can easily make an interpreter that is analogous to the operational semantics of a language.

Consider the rules of $Alangauge$

$$\frac{}{A; n \Rightarrow n}$$

$$\frac{A; e_1 \Rightarrow n_1 \qquad A; e_2 \Rightarrow n_2 \qquad n_3 \text{ is } n_1 + n_2}{A; e_1 + e_2 \Rightarrow n_3}$$

If we consider the premises and the final result of each rule, we can model an interpreter to do exactly what we specify in the rules.

Assuming we have a lexer and parser implemented and a type for both Numbers and an Add construct, we can write an interpreter that follows the above rules:

```
1 def rec eval expr env= match expr with
2  Num(x) -> x
3 |Add(e1,e2) -> let n1 = eval e1 env in
4                let n2 = eval e2 env in
5                let n3 = n1 + n2 in
6                n3;;
```

In moving to $BLanguage$, we gain more rules:

$$\frac{A(x) \Rightarrow v}{A; x \Rightarrow v}$$

$$\frac{A; e_1 \Rightarrow v \qquad A, x : v; e_2 \Rightarrow e_3}{\text{let } x = e_1 \text{ in } e_2 \Rightarrow e_3}$$

We can then update our interpreter accordingly:

```
def rec eval expr env = match expr with
 Num(x) -> x
|Add(e1,e2) ->   let n1 = eval e1 env in
                 let n2 = eval e2 env in
                 let n3 = n1 + n2 in
                 n3
|Var(x) ->       let v = lookup env x in
                 v
|Let(x,e1,e2) -> let v = eval e1 env in
                 let env' = update env (x,v) in
                 let e3 = eval e2 env' in
                 e3;;
```

This assumes that we have an function that adds variable and value pairs to the environment and a function that looks up a variable in the environment.