Operational Semantics

CMSC 330, Spring 2011

1 Introduction

So far in class, we've talked about different ways of specifying the *syntax* of programs, notably regular expressions and context-free grammars. In this lecture, we will examine ways to describe the *semantics* of programs, i.e., what they mean.

Many of today's languages, such as C, C++, and Java, have semantics that are described by long, English-language documents. While this documentation is precise in some sense, it can be hard to know whether the semantics are

- *consistent*—are there places in the language description that conflict with each other?;
- *sufficient*—is every possible valid program assigned a meaning? For example, C leaves many aspects of the language undefined.; and
- *correct*—does the language do what programmers would expect?

The situation is even worse with other languages, such as Ruby, Python, and Perl, that have no clear specification other than their implementation.

2 Kinds of Formal Semantics

In contrast, we are going to study how to write down *formal semantics* for a programming language. While formality doesn't directly address consistency, sufficiency, and correctness, it gives us a much better shot at them. Formal semantics are concise, precise, and subject to formal reasoning, and so it's a lot easier to be sure they're right; and if they're not right, it's easier to find the problems.

There are three main approaches for describing formal semantics:

- An *axiomatic semantics* of a program talks about how logical properties of the program state change as a program runs. For example, if we know that currently $x \geq y \wedge z \geq w$, and the next statement in the program is $a := x + z$[1], then after that statement we also know that $a \geq y + w$. Axiomatic semantics fell out of favor for a long time, but has seen recent re-popularity for tools that perform analysis of program source code. However, for purposes of this class, we will not discuss them further, since they are still hard to use; for example, explaining pointer-ful programs using axiomatic semantics is an area of current research interest.
- A *denotational semantics* of a program "compiles" programs into equivalent mathematical function over something called a *domain*. Denotational semantics was developed by Dana Scott and Christopher Strachey, and was highly influential. However, denotational semantics requires sophisticated mathematical reasoning, and it can be hard to use in practice.
- An *operational semantics* of a program is essentially a program *interpreter*, written out in mathematical notation. Operational semantics is by far the most common way of describes semantics today, and it is what we will focus on in this lecture.

[1] Here := is just ordinary assignment, written in a Pascal-style notation as is traditional in these semantics

3 A Big-Step Operational Semantics

Language. We will define an operational semantics for the OCaml-like language given by the following grammar:

$$e ::= x \mid n \mid\mid \texttt{true} \mid \texttt{false} \mid e + e \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \mid \texttt{fun } x \to e \mid e\ e$$

Here x represents an arbitrary variable, n is an integer, `true` and `false` are booleans, and `if` e_1 `then` e_2 `else` e_3 evaluates to e_2 if e_1 evaluates to `true`, and to e_2 if e_1 evaluates to `false`. Anonymous functions are created with `fun` $x \to e$, and the expression $e_1\ e_2$ calls function e_1 with argument e_2.

In OCaml terms, we can represent ASTs for the above language as

```
type expr =
    EVar of string
  | ENum of int
  | ETrue
  | EFalse
  | EPlus of expr * expr
  | EIf of expr * expr * expr
  | EFun of string * expr
  | EApp of expr * expr
```

where we use `strings` for the names of variables.

Values and environments.

$$\begin{array}{rcl} v & ::= & \mathsf{n} \mid \mathsf{true} \mid \mathsf{false} \mid (A, \lambda x.e) \\ A & ::= & \emptyset \mid x : v, A \end{array}$$

```
type value =
    VNum of int
  | VTrue
  | VFalse
  | VClosure of asst * string * expr
and asst = (string * value) list
```

Semantics rules.

$$\textsc{Int}\ \frac{}{A \vdash n \longrightarrow \mathsf{n}} \qquad \textsc{True}\ \frac{}{A \vdash \texttt{true} \longrightarrow \mathsf{true}} \qquad \textsc{False}\ \frac{}{A \vdash \texttt{false} \longrightarrow \mathsf{false}} \qquad \textsc{Var}\ \frac{}{A \vdash x \longrightarrow A(x)}\ x \in dom(A)$$

$$\textsc{Plus}\ \frac{A \vdash e_1 \longrightarrow \mathsf{n} \qquad A \vdash e_2 \longrightarrow \mathsf{m} \qquad \mathsf{p} = \mathsf{n}+\mathsf{m}}{A \vdash e_1 + e_2 \longrightarrow \mathsf{p}}$$

$$\textsc{If-T}\ \frac{A \vdash e_1 \longrightarrow \texttt{true} \qquad A \vdash e_2 \longrightarrow v}{A \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \longrightarrow v} \qquad \textsc{If-F}\ \frac{A \vdash e_1 \longrightarrow \texttt{false} \qquad A \vdash e_3 \longrightarrow v}{A \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \longrightarrow v}$$

$$\textsc{Fun}\ \frac{}{A \vdash \texttt{fun } x \to e \longrightarrow (A, \lambda x.e)} \qquad \textsc{App}\ \frac{A \vdash e_1 \longrightarrow (A', \lambda x.e) \qquad A \vdash e_2 \longrightarrow v_2 \qquad x : v_2, A' \vdash e \longrightarrow v}{A \vdash e_1\ e_2 \longrightarrow v}$$

```
let rec eval a = function
    EVar x -> List.assoc x a
  | ENum n -> VNum n
  | ETrue -> VTrue
  | EFalse -> VFalse
  | EIf(e1, e2, e3) ->
      begin
match eval a e1 with
    VTrue -> eval a e2
  | VFalse -> eval a e3
      end
  | EFun(x, e) -> VClosure(a, x, e)
  | EApp(e1, e2) ->
      let VClosure(a', x, e) = eval a e1 in
      let v2 = eval a e2 in
eval ((x,v2)::a') e
```

Examples.

```
eval [] (EPlus (ENum 3, ENum 4))
eval [] (EIf(ETrue, ENum 3, ENum 4))
eval [] (EIf(ETrue, (EPlus (ENum 3, ENum 4)), ENum 5))
eval ["x", VNum 3] (EPlus (EVar "x", ENum 4))
eval [] (EApp(EFun("x", EPlus (EVar "x", ENum 4)), ENum 3))
eval [] (EApp(EApp(EFun("x", EFun("y", EPlus(EVar "x", EVar "y"))),
ENum 3), ENum 4))
```

4 A Small-Step Operational Semantics

Expressions and values.

$$\begin{array}{rcl} e & ::= & v \mid x \mid e + e \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \mid e\ e \\ v & ::= & \mathsf{n} \mid \mathsf{true} \mid \mathsf{false} \mid \texttt{fun } x \to e \end{array}$$

Notice that values are just a subset of the expressions, so we won't represent them with a different type in OCaml. We also don't need environments, because we'll apply functions to arguments using substitution.

```
let is_value = function
    ENum _ | ETrue | EFalse | EFun _ -> true
  | _ -> false
```

Substitution.

$$\begin{array}{rcll} \mathsf{n}[x \mapsto v] & = & \mathsf{n} & \\ \mathsf{true}[x \mapsto v] & = & \mathsf{true} & \\ \mathsf{false}[x \mapsto v] & = & \mathsf{false} & \\ x[x \mapsto v] & = & v & \\ y[x \mapsto v] & = & y & x \neq y \\ (e_1 + e_2)[x \mapsto v] & = & (e_1[x \mapsto v]) + (e_2[x \mapsto v]) & \\ (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3)[x \mapsto v] & = & \texttt{if } (e_1[x \mapsto v]) \texttt{ then } (e_2[x \mapsto v]) \texttt{ else } (e_3[x \mapsto v]) & \\ (\texttt{fun } y \to e)[x \mapsto v] & = & \texttt{fun } y \to (e[x \mapsto v]) & x \neq y \\ (\texttt{fun } x \to e)[x \mapsto v] & = & \texttt{fun } x \to e & \\ (e_1\ e_2)[x \mapsto v] & = & (e_1[x \mapsto v])\ (e_2[x \mapsto v]) & \end{array}$$