

CS6848 - Principles of Programming Languages

Principles of Programming Languages

V. Krishna Nandivada

IIT Madras



Outline

- 1 Program Semantics
 - Introduction
 - Big Step Semantics
 - Small Step Semantics



Last class

Interpreters

- A Environment
- B Cells
- C Closures
- D Recursive environments
- E Interpreting OO (MicroJava) programs.



Introduction

- An interpreter executes a program as per the semantics.
- An interpreter can be viewed as an executable description of the semantics of a programming language.
- Program semantics is the field concerned with the rigorous mathematical study of the meaning of programming languages and models of computation.
- Formal ways of describing the programming semantics.
 - Operational semantics - execution of programs in the language is described directly (in the context of an abstract machine).
 - Big-step semantics (with environments) - is close in spirit to the interpreters we have seen earlier.
 - Small-step semantics (with syntactic substitution) - formalizes the inlining of a procedure call as an approach to computation.
 - Denotational Semantics - each phrase in the language is *translated* to a *denotation* - a phrase in some other language.
 - Axiomatic semantics - gives meaning to phrases by describing the logical axioms that apply to them.



- The traditional syntax for procedures in the lambda-calculus uses the Greek letter λ (lambda), and the grammar for the lambda-calculus can be written as:

$$e ::= x \mid \lambda x.e \mid e_1 e_2$$

$$x \in \text{Identifier (infinite set of variables)}$$

- Brackets are only used for grouping of expressions. Convention for saving brackets:
 - that the body of a λ -abstraction extends “as far as possible.”
 - For example, $\lambda x.xy$ is short for $\lambda x.(xy)$ and not $(\lambda x.x)y$.
 - Moreover, $e_1 e_2 e_3$ is short for $(e_1 e_2)e_3$ and not $e_1(e_2 e_3)$.



Outline

1 Program Semantics

- Introduction
- Big Step Semantics
- Small Step Semantics



We will give the semantics for the following extension of the lambda-calculus:

$$e ::= x \mid \lambda x.e \mid e_1 e_2 \mid c \mid \text{succ } e$$

$$x \in \text{Identifier (infinite set of variables)}$$

$$c \in \text{Integer}$$



Big step semantics

Here is a big-step semantics with environments for the lambda-calculus.

$$w, v \in \text{Value}$$

$$v ::= c \mid (\lambda x.e, \rho)$$

$$\rho \in \text{Environment}$$

$$\rho ::= x_1 \mapsto v_1, \dots, x_n \mapsto v_n$$

The semantics is given by the following five rules:

$$(1) \quad \rho \vdash x \triangleright v \quad (\rho(x) = v)$$

$$(2) \quad \rho \vdash \lambda x.e \triangleright (\lambda x.e, \rho)$$

$$(3) \quad \frac{\rho \vdash e_1 \triangleright (\lambda x.e, \rho') \quad \rho \vdash e_2 \triangleright v \quad \rho', x \mapsto v \vdash e \triangleright w}{\rho \vdash e_1 e_2 \triangleright w}$$

$$(4) \quad \rho \vdash c \triangleright c$$

$$(5) \quad \frac{\rho \vdash e \triangleright c_1}{\rho \vdash \text{succ } e \triangleright c_2} \quad [c_2] = [c_1] + 1$$



1 Program Semantics

- Introduction
- Big Step Semantics
- Small Step Semantics



- In small step semantics, one step of computation = either one primitive operation, or inline one procedure call.
- We can do steps of computation in different orders:

```
> (define foo
    (lambda (x y) (+ (* x 3) y)))
> (foo (+ 4 1) 7)
22
```

Let us calculate:

```
(foo (+ 4 1) 7)
=> ((lambda (x y) (+ (* x 3) y))
    (+ 4 1) 7)
=> (+ (* (+ 4 1) 3) 7)
=> 22
```



Small step semantics (contd.)

We can also calculate like this:

```
(foo
 (+ 4 1) 7)
=> (foo 5 7)
=> ((lambda (x y) (+ (* x 3) y))
    5 7)
=> (+ (* 5 3) 7)
=> 22
```



Free variables

A variable x occurs *free* in an expression E iff x is not bound in E . Examples:

- no variables occur free in the expression
- the variable y occurs free in the expression

```
(lambda (y) ((lambda (x) x) y))
```

```
((lambda (x) x) y)
```

An expression is *closed* if it does not contain free variables.
A program is a closed expression.



Methods of procedure application

Call by value

```
((lambda (x) x)
  ((lambda (y) (+ y 9)) 5))
```

=> ((lambda (x) x) (+ 5 9))

=> ((lambda (x) x) 14)

=> 14

Always evaluate the arguments first

- Example: Scheme, ML, C, C++, Java



Methods of procedure application

Call by name (or lazy-evaluation)

```
((lambda (x) x)
  ((lambda (y) (+ y 9)) 5))
```

=> ((lambda (y) (+ y 9)) 5)

=> (+ 5 9)

=> 14

Avoid the work if you can

- Example: Miranda and Haskell

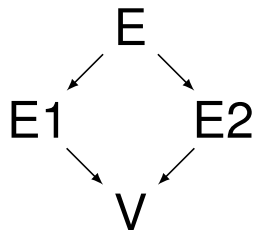
Lazy or eager: Is one more efficient? Are both the same?



Difference

- Q: If we run the same program using these two semantics, can we get different results?
- A:
 - If the run with call-by-value reduction terminates, then the run with call-by-name reduction terminates. (But the converse is in general false).
 - If both runs terminate, then they give the same result.

Church Rosser theorem



Call by value - too eager?

Sometimes call-by-value reduction fails to terminate, even though call-by-name reduction terminates.

```
(define delta (lambda (x) (x x)))
(delta delta)
=> (delta delta)
=> (delta delta)
=> ...
```

Consider the program:

```
(define const (lambda (y) 7))
(const (delta delta))
```

- call by value reduction fails to terminate; cannot finish evaluating the operand.
- call by name reduction terminates.



Summary - calling convention

- call by value is more efficient but may not terminate
- call by name may evaluate the same expression multiple times.
- Lazy languages uses - call-by-need.
- Languages like Scala allow both call by value and name!



Notes on reduction

- Applicative order reduction - A β reduction can be applied only if both the operator and the operand are already values. Else?
- Applicative order reduction (call by value), example: Scheme, C, Java.



Beta reduction

- A procedure call which is ready to be “inlined” is called a *beta-redex*. Example $(\lambda \text{var} \text{ body } \text{rand})$
- In lambda-calculus call-by-value and call-by-name reduction allow the choosing of arbitrary beta-redex.
- The process of inlining a beta-redex for some reducible expression is called *beta-reduction*.

$(\lambda \text{var} \text{ body } \text{rand}) \text{ body}[\text{var}:=\text{rand}]$

- η conversion: A simple optimization:

$$(\lambda x (E x)) = E$$

- A *conversion* when applied in the left-to-right direction is called a *reduction*.



Notes on reduction

- Is there a reduction strategy which is guaranteed to find the answer if it exists? – *leftmost* reduction (lazy evaluation).
- leftmost-reduction – reduce the β -redex whose left parenthesis comes first
- A lambda expression is in *normal* form if it contains no β -redexes.
- An expression in normal form – cannot be further reduced. e.g. constant or $(\lambda x) x$
- Church-Rosser theorem \rightarrow expression can have at most one normal form.
- leftmost reduction will find the normal form of an expression if one exists.



Name clashes

- Care must be taken to avoid name clashes. Example:

```
((lambda (x)
  (lambda (y) (y x)))
 (y 5))
```

should not be transformed into

```
(lambda (y) (y (y 5)))
```

- The reference to `y` in `(y 5)` should remain **free**!
- The solution is to change the name of the inner variable name `y` to some name, say `z`, that does not occur free in the argument `y 5`.

```
((lambda (x)
  (lambda (z) (z x)))
 (y 5))
```

=> (lambda (z) (z (y x))) ; ; y is free.



Substitution

- The notation $e[x := M]$ denotes e with M substituted for every free occurrence of x in such a way that name clashes are avoided.

- We will define $e[x := M]$ inductively on e .

$$\begin{aligned} x[x := M] &\equiv M \\ y[x := M] &\equiv y \quad (x \neq y) \\ (\lambda x. e_1)[x := M] &\equiv (\lambda x. e_1) \\ (\lambda y. e_1)[x := M] &\equiv \lambda z. ((e_1[y := z])[x := M]) \\ &\quad \text{(where } x \neq y \text{ and } z \text{ does not} \\ &\quad \text{occur free in } e_1 \text{ or } M\text{).} \end{aligned}$$

$$\begin{aligned} (e_1 e_2)[x := M] &\equiv (e_1[x := M])(e_2[x := M]) \\ c[x := M] &\equiv c \\ (succ\ e_1)[x := M] &\equiv succ\ (e_1[x := M]) \end{aligned}$$

- The renaming of a bound variable by a *fresh* variable is called *alpha-conversion*.
- Q: Can we avoid creating a new variable in the fourth rule ?



Small step semantics

Here is a small-step semantics with syntactic substitution for the lambda-calculus.

$$\begin{aligned} v &\in \text{Value} \\ v &::= c \mid \lambda x. e \end{aligned}$$

The semantics is given by the reflexive, transitive closure of the relation \rightarrow_v

$$\rightarrow_v \subseteq \text{Expression} \times \text{Expression}$$

(6) $\lambda x. e \rightarrow_v \lambda x. e[x := v]$

(7)
$$\frac{e_1 \rightarrow_v e'_1}{e_1 e_2 \rightarrow_v e'_1 e_2}$$

(8)
$$\frac{e_2 \rightarrow_v e'_2}{ve_2 \rightarrow_v ve'_2}$$

(9) $succ\ c_1 \rightarrow_v succ\ c_2 \quad ([c_2] = [c_1] + 1)$

(10)
$$\frac{e_1 \rightarrow_v e_2}{succ\ e_1 \rightarrow_v succ\ e_2}$$

