Operational Semantics

CS 565
9/20/11

Review

- Abstract syntax trees
  - view as parse tree for a program
    - independent of concrete syntax
    - does not provide a semantics for operators
  - BNF grammars and related inductive definition styles allow us to:
    - specify certain structural properties of programs (e.g., size, depth, etc.) without knowledge of their semantics
    - write inductive-style proofs that relate these properties

Semantics

- We are ultimately interested in the meaning of programs:
  - How do we define "meaning"?
  - How do we understand notions like "evaluation", "compilation", etc.?
  - How is "evaluation" related to "meaning"?
  - How do we capture notions like "non-termination", "recursion", etc. in defining the "meaning" of a program?

Abstract machines

First approach: define an abstract machine.

The behavior of the machine on a program defines the program's "meaning".

- An abstract machine consists of:
  - a set of states
  - a transition relation on states (→)
- Evaluation stops when we reach a state in which no further transitions are possible.

States and Transitions

- States record all salient information in a machine:
  - program counter
  - register contents
  - memory
  - code
- In studying languages, we can abstract these complex low-level structures to simpler high-level ones
  - For the simple language of arithmetic, the state is simply the term being evaluated
- The transition relation is often a partial function on states:
  - Not all states have a transition
- If a state does have a transition, the resulting state is unique
  - determinacy of evaluation

Booleans

Syntax of terms and values

t ::=		terms
	`true`	constant true
|	`false`	constant false
|	`if` t `then` t `else` t	conditional
v ::=		values
	`true`	true value
|	`false`	false value

Transition (Evaluation) Relation

The relation $t \rightarrow t'$ is the smallest relation closed under the following rules:

- `if true then` t_2 `else` $t_3 \rightarrow t_2$
- `if false then` t_2 `else` $t_3 \rightarrow t_3$
- $$\frac{t_1 \rightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t2 \texttt{ else } t_3 \rightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}$$

Terminology

- Computation rules
  - if true then t_2 else $t_3 \rightarrow t_2$
  - if false then t_2 else $t_3 \rightarrow t_3$
- Congruence rule
  - $$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3}$$

Computation rules perform "real" computation steps.

Congruence rules guide evaluation order; they determine where computation rules can be next applied

Example

Consider a different evaluation strategy such that

(a) the then and else branches are evaluated (in that order) before the guard and

(b) if the then and else branches both yield the same value, we omit evaluation of the guard.

How would we write this evaluator?

An alternative evaluator

- if true then v_t else $v_f \rightarrow v_t$
- if false then v_t else $v_f \rightarrow v_f$
- if t_1 then v else $v \rightarrow v$
- $$\frac{t_2 \rightarrow t_2'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1 \text{ then } t_2' \text{ else } t_3}$$
- $$\frac{t_3 \rightarrow t_3'}{\text{if } t_1 \text{ then } v_t \text{ else } t_3 \rightarrow \text{if } t_1 \text{ then } v_t \text{ else } t_3'}$$
- $$\frac{t_1 \rightarrow t_1'}{\text{if } t_1 \text{ then } v_t \text{ else } v_f \rightarrow \text{if } t_1' \text{ then } v_t \text{ else } v_f}$$

Induction

We view the transition relation as the smallest binary relation on terms satisfying the rules. If $(t, t') \in \rightarrow$, then the judgment $t \rightarrow t'$ is derivable.

A derivation tree is a tree whose leaves are instances of computation rules (e.g., true and false transitions) and whose internal nodes are congruence rules.

This notion of evaluation as a construction of a tree leads to an inductive proof technique on induction on derivations.

Derivation trees

Consider the following terms:

s ≡ `if true then false else false`

t ≡ `if` **s** `then true else true`

u ≡ `if false then true else true`

What is the derivation tree for the judgment?

`if` **t** `then false else false` →

`if` **u** `then false else false`

Derivation Trees

$$\dfrac{\dfrac{}{s \to \texttt{false}}}{t \to u}$$

$$\texttt{if } t \texttt{ then false else false} \to \texttt{if } u \texttt{ then false else false}$$

Induction

Theorem: if $t \rightarrow t'$ and $t \rightarrow t''$ then $t' = t''$.

Proof: By induction on the derivation of $t \rightarrow t'$. At each step of the induction, assume theorem holds for all smaller derivations. Proceed by case analysis of the evaluation rule used at the root of the derivation.

Theorem: if $t \rightarrow t'$ then size(t) > size(t')

Normal forms

- A term t is in normal form if no evaluation rule applies to it, i.e., there is no t′ such that $t \rightarrow t'$.
- Every value is in normal form.
- Theorem: Every term that is in normal form is a value.
  - Proof: How would you prove this?

Normal forms

Theorem: Every term t that is in normal form is a value.

Proof: By structural induction on t and contradiction.

Suppose t is not a value. Then t must have the form "if t_1 then t_2 else t_3." Now, t_1 can be either true or false in which case t is not in normal form (there is a computation rule that matches), or t_1 is another if expression. By the induction hypothesis, t_1 is not in normal form, hence t is not in normal form.

Normal forms

- Is it always the case for real languages that a term which is in normal form is always a value?
  - In real languages normal forms may also correspond to expressions that are ill-typed or which correspond to runtime errors.
    - E.g., true + 3 → ??? or

      succ false → ?

      These terms are in normal form (why?) but do not correspond to values as defined by the machine specification.
  - A term is said to be stuck if it is normal form but is not a value

IMP: A simple imperative language

Syntactic categories:

- int — integers
  - $n \in Z$
- bool — boolean
  - `true, false`
- L — locations
  - x,y,…
- Aexp — Arithmetic expressions
  - e
- Bexp — Boolean expressions
  - b
- Com — Commands
  - c
- Values
  - v ::= n | `true` | `false`

Abstract syntax (AExp)

Arithmetic expressions:

```
e ::=   n
      | x
      | e_1 + e_2
      | e_1 - e_2
      | e_1 * e_2
```

- Variables are used directly in expressions (no prior declaration)
- All variables are presumed to have integer type
- No side-effects (e.g., overflow, etc.)

Abstract Syntax (BExp)

Boolean expressions:

$b ::=$ `true`
$\mid$ `false`
$\mid e_1 = e_2$
$\mid e_1 \leq e_2$
$\mid \neg b$
$\mid b_1 \wedge b_2$
$\mid b_1 \vee b_2$

Abstract syntax (Comm)

Commands

c ::= `skip`
| x := e
| c_1; c_2
| `if` b `then` c_1 `else` c_2
| `while` b `do` c

- Typing rules expressed implicitly in the choice of meta-variables
- All side-effects captured within commands
- Do not consider functions, pointers, data structures

Operational Semantics for IMP

- Unlike the simple language of booleans and conditionals or arithmetic, IMP programs bind variables to locations, and can side-effect the contents of these locations.
- Define $\sigma \in \Sigma = L \rightarrow Z$ to define the state of program memory.
- Evaluation judgements take one of the following forms:
  - $c, \sigma \rightarrow \sigma'$
  - $t, \sigma \rightarrow t', \sigma'$
    - $t \in Term = Aexp + Bexp + Com + Value$

Semantics for Aexp

$$x,\sigma \rightarrow \sigma(x),\sigma$$

$$\frac{e_1,\sigma \rightarrow e_1',\sigma}{e_1 + e_2,\sigma \rightarrow e_1' + e_2,\sigma}$$

$$\frac{e_2,\sigma \rightarrow e_2',\sigma}{n_1 + e_2,\sigma \rightarrow n_1 + e_2',\sigma}$$

Note:

- σ does not change; this is because arithmetic expressions do not have side-effects
- distinctions between normal forms (values) and expressions expressed in the choice of meta-variables used in the rules
- order of evaluation expressed in the definition of the rules
- similar operations apply for the remaining arithmetic operations

Semantics for BExp

$\texttt{true}, \sigma \rightarrow \texttt{true}, \sigma$

$\texttt{false}, \sigma \rightarrow \texttt{false}, \sigma$

$$\frac{e_1, \sigma \rightarrow e_1', \sigma}{e_1 = e_2, \sigma \rightarrow e_1' = e_2, \sigma}$$

$$\frac{e_2, \sigma \rightarrow e_2', \sigma}{\texttt{true} = e_2, \sigma \rightarrow \texttt{true} = e_2', \sigma}$$

Semantics for Com

$\texttt{skip};c, \sigma \rightarrow c, \sigma$

$\texttt{skip}, \sigma \rightarrow \sigma$

$$\frac{e, \sigma \rightarrow e', \sigma'}{x := e, \sigma \rightarrow x := e', \sigma'}$$

$x := v; c, \sigma \rightarrow c, \sigma[x \mapsto v]$

$x := v, \sigma \rightarrow \sigma[x \mapsto v]$

$\texttt{while}$ false $\texttt{do}$ $c;c', \sigma \rightarrow c', \sigma$

$\texttt{while}$ false $\texttt{do}$ $c, \sigma \rightarrow \sigma$

$$\frac{b, \sigma \rightarrow \text{true}, \sigma}{\texttt{while}\ b\ \texttt{do}\ c, \sigma \rightarrow c; \texttt{while}\ b\ \texttt{do}\ c, \sigma}$$

$$\frac{b, \sigma \rightarrow b', \sigma}{\texttt{while}\ b\ \texttt{do}\ c, \sigma \rightarrow \texttt{while}\ b'\ \texttt{do}\ c, \sigma}$$

What's wrong with these rules? Hint: consider substitution from b to b'. Suppose b refers to locations modified by c. How would you fix it?

Next time

- Will look at another way of defining an operational semantics:
  - Natural or "big-step" semantics
- Transplanting these ideas to Coq