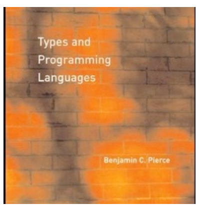
Operational Semantics

CS 565

Lecture 3



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"?
- 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 ($\rightarrow$)

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

Booleans

Syntax of terms and values

$$
\begin{array}{llll}
t & ::= & & \\
 & | & \texttt{if } t \texttt{ then } t' \texttt{ else } t'' & \text{conditional} \\
 & | & \texttt{true} & \text{true constant} \\
 & | & \texttt{false} & \text{false constant} \\
 & & & \\
v & ::= & & \\
 & | & \mathbf{tt} & \text{true value} \\
 & | & \mathbf{ff} & \text{false value}
\end{array}
$$

Transition (Evaluation) Relation

The relation t → t' is the smallest relation closed under the following rules:

$\boxed{t_1 \longrightarrow t_2}$

$$\frac{}{\texttt{if } t \texttt{ then } t' \texttt{ else } t'' \longrightarrow t_1} \quad \text{RTRUE}$$

$$\frac{}{\texttt{if } t \texttt{ then } t' \texttt{ else } t'' \longrightarrow t_2} \quad \text{RFALSE}$$

$$\frac{t \longrightarrow t'}{\texttt{if } t \texttt{ then } t' \texttt{ else } t'' \longrightarrow \texttt{if } t \texttt{ then } t' \texttt{ else } t''} \quad \text{RRED}$$

Terminology

Computation rules

$$\frac{}{\texttt{if } t \texttt{ then } t' \texttt{ else } t'' \longrightarrow t_1} \quad \text{RTRUE}$$

$$\frac{}{\texttt{if } t \texttt{ then } t' \texttt{ else } t'' \longrightarrow t_2} \quad \text{RFALSE}$$

Congruence rule

$$\frac{t \longrightarrow t'}{\texttt{if } t \texttt{ then } t' \texttt{ else } t'' \longrightarrow \texttt{if } t \texttt{ then } t' \texttt{ else } t''} \quad \text{RRED}$$

Computation rules perform “real” computation steps.

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

Ott definition

```
grammar
t  ::  't_' ::=
 | if t then t' else t'' :: :: IfThen  {{com conditional}}
 | true                  :: :: True    {{ com true constant}}
 | false                 :: :: False   {{ com false constant }}

defns R :: '' ::=
defn  t1 --> t2 :: :: reduce :: ''
by
-------------------------------        :: Rtrue
if true then t1 else t2 --> t1

-------------------------------        :: Rfalse
if false then t1 else t2 --> t2
t --> t'
---------------------------------------------  :: Rred
if t then t1 else t2 --> if t' then t1 else t2
```

Example

Consider a different evaluation strategy such that

- the then and else branches are evaluated (in that order) before the guard and
- 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 vt else vf → vt

if false then vt else vf → vf

if t1 then v else v → v

$$\boxed{t_1 \;\rightarrow\; t_2}$$

$$\frac{}{\texttt{if}\ t\ \texttt{then}\ t'\ \texttt{else}\ t'' \;\rightarrow\; t_1}\ \textsc{Rtrue}$$

$$\frac{}{\texttt{if}\ t\ \texttt{then}\ t'\ \texttt{else}\ t'' \;\rightarrow\; t_2}\ \textsc{Rfalse}$$

$$\frac{t_1 \;\rightarrow\; t'}{\texttt{if}\ t\ \texttt{then}\ t'\ \texttt{else}\ t'' \;\rightarrow\; \texttt{if}\ t\ \texttt{then}\ t'\ \texttt{else}\ t''}\ \textsc{RredC}$$

$$\frac{t_2 \;\rightarrow\; t'}{\texttt{if}\ t\ \texttt{then}\ t'\ \texttt{else}\ t'' \;\rightarrow\; \texttt{if}\ t\ \texttt{then}\ t'\ \texttt{else}\ t''}\ \textsc{RredT}$$

$$\frac{t_3 \;\rightarrow\; t'}{\texttt{if}\ t\ \texttt{then}\ t'\ \texttt{else}\ t'' \;\rightarrow\; \texttt{if}\ t\ \texttt{then}\ t'\ \texttt{else}\ t''}\ \textsc{RredF}$$

Induction

We view the transition relation as the smallest binary relation on terms satisfying the rules. If (`t`,`t`') ∈ →, then the judgment `t` → `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

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

Induction

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

Proof: By induction on the derivation of $t \to t'$. 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 $t \to t'$ 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 $t \to t'$.

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.
- `t` must have the form "`if t1 then t2 else t3`"

  Now, `t1` can be either `true` or `false` in which case `t` is not in normal form (there is a computation rule that matches), or `t1` is another if expression.

  By the induction hypothesis (*), `t1` 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
- bool — Boolean
- loc — Locations
- Aexp — Arithmetic expressions
- Bexp — Boolean expressions
- Com — Commands
- Values

  v ::= n | true | false

Abstract syntax (AExp)

Arithmetic expressions:

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

$$
\begin{array}{lll}
a & ::= & \\
 & | & int \\
 & | & \mathbf{x} \\
 & | & a_1 + a_2 \\
 & | & a_1 * a_2 \\
 & | & a_1 - a_2
\end{array}
$$

Abstract Syntax (BExp)

Boolean expressions:

$$
\begin{array}{lcl}
b & ::= & \\
 & \mid & bool \\
 & \mid & e_1 = e_2 \\
 & \mid & e_1 \prec e_2 \\
 & \mid & \textbf{not}\ b \\
 & \mid & b_1\ \textbf{and}\ b_2 \\
 & \mid & b_1\ \textbf{or}\ b_2
\end{array}
$$

Abstract syntax (Comm)

Commands

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

$$
\begin{array}{lcl}
c & ::= & \\
 & \mid & \textbf{skip} \\
 & \mid & \mathbf{x} := a \\
 & \mid & c_1\ ;\ c_2 \\
 & \mid & \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \\
 & \mid & \textbf{while}\ b\ \textbf{do}\ c
\end{array}
$$

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, $\sigma \rightarrow$ c, σ'
- e, $\sigma \rightarrow$ e'

  e $\in$ exp = Aexp + Bexp + Com + Value

Semantics for Aexp

Notes

- σ does not change; because aexps 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

$$\frac{\sigma\,(\mathbf{x}) = \mathit{int}}{\mathbf{x},\, \sigma \longrightarrow \mathit{int}} \quad \textsc{AexpVar}$$

$$\frac{a_1,\, \sigma \longrightarrow a_1'}{a_1 + a_2,\, \sigma \longrightarrow a_1' + a_2} \quad \textsc{AexpPlusL}$$

$$\frac{a_2,\, \sigma \longrightarrow a_2'}{\mathit{int} + a_2,\, \sigma \longrightarrow \mathit{int} + a_2'} \quad \textsc{AexpPlusR}$$

$$\frac{\mathit{int}_1 + \mathit{int}_2 = \mathit{int}_3}{\mathit{int}_1 + \mathit{int}_2,\, \sigma \longrightarrow \mathit{int}_3} \quad \textsc{AexpPlus}$$

Semantics for Aexp

$$\frac{a_1,\ \sigma \longrightarrow a_1'}{a_1 * a_2,\ \sigma \longrightarrow a_1' * a_2}\quad \textsc{AexpTimesL}$$

$$\frac{a_2,\ \sigma \longrightarrow a_2'}{int * a_2,\ \sigma \longrightarrow int * a_2'}\quad \textsc{AexpTimesR}$$

$$\frac{int_1 * int_2 = int_3}{int_1 * int_2,\ \sigma \longrightarrow int_3}\quad \textsc{AexpTimes}$$

$$\frac{a_1,\ \sigma \longrightarrow a_1'}{a_1 - a_2,\ \sigma \longrightarrow a_1' - a_2}\quad \textsc{AexpSubL}$$

$$\frac{a_2,\ \sigma \longrightarrow a_2'}{int - a_2,\ \sigma \longrightarrow int - a_2'}\quad \textsc{AexpSubR}$$

$$\frac{int_1 - int_2 = int_3}{int_1 - int_2,\ \sigma \longrightarrow int_3}\quad \textsc{AexpSub}$$

Semantics for BExp

$$\frac{}{v = v,\ \sigma \longrightarrow \texttt{true}}\quad \textsc{BexpEq}$$

$$\frac{v \neq v'}{v = v',\ \sigma \longrightarrow \texttt{false}}\quad \textsc{BexpNEq}$$

$$\frac{e_1,\ \sigma \longrightarrow e_1'}{e_1 = e_2,\ \sigma \longrightarrow e_1' = e_2}\quad \textsc{BexpEqL}$$

$$\frac{e_2,\ \sigma \longrightarrow e_2'}{int = e_2,\ \sigma \longrightarrow int = e_2'}\quad \textsc{BexpEqR}$$

$$\frac{}{\textbf{not}\ \texttt{true},\ \sigma \longrightarrow \texttt{false}}\quad \textsc{BexpNotT}$$

$$\frac{}{\textbf{not}\ \texttt{false},\ \sigma \longrightarrow \texttt{true}}\quad \textsc{BexpNotF}$$

$$\frac{b,\ \sigma \longrightarrow b'}{\textbf{not}\ b,\ \sigma \longrightarrow \textbf{not}\ b'}\quad \textsc{BexpNot}$$

$$\frac{bool_1\ \textbf{and}\ bool_2 = bool}{bool_1\ \textbf{and}\ bool_2,\ \sigma \longrightarrow bool}\quad \textsc{BexpAnd}$$

$$\frac{b_1,\ \sigma \longrightarrow b_1'}{b_1\ \textbf{and}\ b_2,\ \sigma \longrightarrow b_1'\ \textbf{and}\ b_2}\quad \textsc{BexpAndL}$$

$$\frac{b_2,\ \sigma \longrightarrow b_2'}{bool\ \textbf{and}\ b_2,\ \sigma \longrightarrow bool\ \textbf{and}\ b_2'}\quad \textsc{BexpAndR}$$

$$\frac{bool_1\ \textbf{or}\ bool_2 = bool}{bool_1\ \textbf{or}\ bool_2,\ \sigma \longrightarrow bool}\quad \textsc{BexpOr}$$

$$\frac{b_1,\ \sigma \longrightarrow b_1'}{b_1\ \textbf{or}\ b_2,\ \sigma \longrightarrow b_1'\ \textbf{or}\ b_2}\quad \textsc{BexpOrL}$$

$$\frac{b_2,\ \sigma \longrightarrow b_2'}{bool\ \textbf{or}\ b_2,\ \sigma \longrightarrow bool\ \textbf{or}\ b_2'}\quad \textsc{BexpOrR}$$

Semantics for Com

$$\frac{}{\mathbf{skip}\,;\ c,\ \sigma\ \longrightarrow\ c,\ \sigma}\ \textsc{Skip}$$

$$\frac{a,\ \sigma\ \longrightarrow\ a'}{\mathbf{x} := a\,;\ c,\ \sigma\ \longrightarrow\ \mathbf{x} := a'\,;\ c,\ \sigma}\ \textsc{AssignE}$$

$$\frac{\sigma = \sigma\,[\mathbf{x}\ \mapsto\ \mathit{int}]}{\mathbf{x} := \mathit{int}\,;\ c,\ \sigma\ \longrightarrow\ c,\ \sigma}\ \textsc{Assign}$$

$$\frac{b,\ \sigma\ \longrightarrow\ b'}{\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\,;\ c_3,\ \sigma\ \longrightarrow\ \mathbf{if}\ b'\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\,;\ c_3,\ \sigma}\ \textsc{IfE}$$

$$\frac{}{\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\,;\ c_3,\ \sigma\ \longrightarrow\ c_1\,;\ c_3,\ \sigma}\ \textsc{IfT}$$

$$\frac{}{\mathbf{if}\ \mathtt{false}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\,;\ c_3,\ \sigma\ \longrightarrow\ c_2\,;\ c_3,\ \sigma}\ \textsc{IfF}$$

$$\frac{b,\ \sigma\ \longrightarrow\ b'}{\mathbf{while}\ b\ \mathbf{do}\ c_1\,;\ c_2,\ \sigma\ \longrightarrow\ \mathbf{while}\ b'\ \mathbf{do}\ c_1\,;\ c_2,\ \sigma}\ \textsc{WhileE}$$

$$\frac{}{\mathbf{while}\ \mathtt{true}\ \mathbf{do}\ c_1\,;\ c_2,\ \sigma\ \longrightarrow\ c_1\,;\ \mathbf{while}\ \mathtt{true}\ \mathbf{do}\ c_1\,;\ c_2,\ \sigma}\ \textsc{WhileT}$$

$$\frac{}{\mathbf{while}\ \mathtt{false}\ \mathbf{do}\ c_1\,;\ c_2,\ \sigma\ \longrightarrow\ c_2,\ \sigma}\ \textsc{WhileF}$$

Semantics for Com

There are some issues with the Com rules:

- There are many “uninteresting” rules that merely reduce subexpressions
- All programs must be terminated by `skip`
- What happens in the `While` rules if `b` depends on state modified by `c1`?

  (Think of a fix)