A Simple Language of Arithmetic Expressions

syntax and operational semantics

Review of ARITH

• Syntax - what programs look like
  
  $e ::= n \quad | \quad e_1 + e_2 \quad | \quad e_1 * e_2$
  
  - integer literals
  - sum
  - product

• Semantics - what programs mean

• Operational Semantics - describe program meaning in terms of run-time behavior
  
  - $e \Downarrow n$

Operational Semantics as Inference Rules

- $e_1 \Downarrow n_1$, $e_2 \Downarrow n_2$ $\Rightarrow$ $n$ is the sum of $n_1$ and $n_2$
  
  $\quad e_1 + e_2 \Downarrow n$

- $e_1 \Downarrow n_1$, $e_2 \Downarrow n_2$ $\Rightarrow$ $n$ is the product of $n_1$ and $n_2$
  
  $\quad e_1 * e_2 \Downarrow n$

• Meaning: “above the line” implies “below the line”

• These rules are:
  - evaluation rules for the big-step operational semantics
  - derivation rules for the judgement $e \Downarrow n$

How to Read the Rules?

• Forward, as inference rules:
  - If we know that the hypothesis judgments hold then we can infer that the conclusion judgment also holds.
  - E.g., if we know that $e_1 \Downarrow 5$ and $e_2 \Downarrow 7$, then we can infer that $e_1 * e_2 \Downarrow 12$.

• Backward, as evaluation rules:
  - Suppose we want to evaluate $e_1 * e_2$
    - i.e., find $n$ s.t. $e_1 * e_2 \Downarrow n$
    - By inspection of the rules we notice that the last step in the derivation of $e_1 * e_2 \Downarrow n$ must be the addition rule: the conclusions of other rules would not match $e_1 * e_2 \Downarrow n$.
    - (This is called reasoning by inversion on the derivation rules.)
    - Thus we must find $n_1$ and $n_2$ such that $e_1 \Downarrow n_1$ and $e_2 \Downarrow n_2$ are derivable. And this is done recursively.
    - Since there is exactly one rule for each kind of expression we say that the rules are syntax-directed.
  - At each step at most one rule applies.
  - This allows a simple evaluation procedure i.e. an algorithm.
OCAML code for ARITH Operational Semantics

```
type exp =  
  | Int of int  
  | Sum of exp * exp  
  | Prod of exp * exp;;

let anexp = Prod(Sum(Int 3, Int 5), Sum(Int 4, Int 2));;

let rec eval e =  
  match e with  
  | Int n -> n  
  | Sum (e1,e2) -> eval(e1) + eval(e2)  
  | Prod (e1,e2) -> eval(e1) * eval(e2);;

eval anexp;;
```

Semantic Properties: Uniqueness of Results

- ARITH is a deterministic language: all expressions evaluate to at most one value.

\[\forall e \in \text{ARITH}, \forall n, n' \in \text{INT} \quad e \Rightarrow n \land e \Rightarrow n' \implies n = n'\]

An Imperative Language: IMP

syntax and operational semantics

a little more interesting

IMP Syntactic Entities

- int integer literals
  - n
- bool booleans
  - p in \{true, false\}
- L locations (assignable variables)
  - x, y, ...
- Aexp arithmetic expressions
  - e
- Bexp boolean expressions
  - b
- Comm commands
  - c

Abstract Syntax (Aexp)

- For arithmetic expressions (Aexp)
  \[ e ::= n \quad \text{for } n \in \text{INT} \]
  \[ | x \quad \text{for } x \in \text{L} \]
  \[ | e_1 + e_2 \quad \text{for } e_1, e_2 \in \text{Aexp} \]
  \[ | e_1 - e_2 \quad \text{for } e_1, e_2 \in \text{Aexp} \]
  \[ | e_1 * e_2 \quad \text{for } e_1, e_2 \in \text{Aexp} \]

- Notes:
  - Variables are not declared.
  - All variables have integer type.
  - There are no side-effects.
Abstract Syntax (Bexp)

• For boolean expressions (Bexp)

\[
\begin{align*}
  b &::= \text{true} \\
  &| \text{false} \\
  &| e_1 = e_2 \text{ for } e_1, e_2 \in \text{Aexp} \\
  &| e_1 < e_2 \text{ for } e_1, e_2 \in \text{Aexp} \\
  &| \neg b \text{ for } b \in \text{Bexp} \\
  &| b_1 \land b_2 \text{ for } b_1, b_2 \in \text{Bexp} \\
  &| b_1 \lor b_2 \text{ for } b_1, b_2 \in \text{Bexp}
\end{align*}
\]

Abstract Syntax (Comm)

• For commands (Comm)

\[
\begin{align*}
  c &::= \text{skip} \\
  &| x := e \text{ for } x \in \text{L and } e \in \text{Aexp} \\
  &| c_1 ; c_2 \text{ for } c_1, c_2 \in \text{Comm} \\
  &| \text{if } b \text{ then } c_1 \text{ else } c_2 \text{ for } c_1, c_2 \in \text{Comm} \text{ and } b \in \text{Bexp} \\
  &| \text{while } b \text{ do } c \text{ for } c \in \text{Comm} \text{ and } b \in \text{Bexp}
\end{align*}
\]

Semantics of IMP

• The meaning of IMP expressions depends on the values of variables.

• A state \( \sigma \) is a function from \( \text{L} \) to \( \text{Z} \)
  - Represents the value of variables at a given moment
  - The set of all states is \( \Sigma = \text{L} \rightarrow \text{Z} \).

Evaluation Rules (for Aexp)

\[
\begin{align*}
  \langle n, \sigma \rangle \cup n \\
  \langle \text{x, } \sigma \rangle \cup \sigma(\text{x}) \\
  \langle e_1 + e_2, \sigma \rangle \cup n_1 + n_2 \\
  \langle e_1 - e_2, \sigma \rangle \cup n_1 - n_2 \\
  \langle e_1 * e_2, \sigma \rangle \cup n_1 * n_2 \\
  \langle e_1, \sigma \rangle \cup n_1 \\
  \langle e_2, \sigma \rangle \cup n_2
\end{align*}
\]

Evaluation Rules (for Bexp)

\[
\begin{align*}
  \langle \text{true}, \sigma \rangle \cup \text{true} \\
  \langle \text{false}, \sigma \rangle \cup \text{false} \\
  \langle e_1 < e_2, \sigma \rangle \cup p \text{ is } n_1 < n_2 \\
  \langle e_1, \sigma \rangle \cup n_1 \\
  \langle e_2, \sigma \rangle \cup n_2 \\
  \langle b_1, \sigma \rangle \cup \text{false} \\
  \langle b_1 \lor b_2, \sigma \rangle \cup \text{false} \\
  \langle b_1 \land b_2, \sigma \rangle \cup \text{true}
\end{align*}
\]

Operational Semantics of IMP

• Evaluation judgment for expressions
  - A ternary relation: on an expression, a state, and a value.
  - We write: \( \langle e, \sigma \rangle \Downarrow n \)
  - The evaluation of expressions does not have side-effects, so no resulting state on the right
  - In this case we can also view this judgment as a function of two arguments (written to the left of \( \Downarrow \))

• Evaluation judgement for commands
  - A ternary relation: on an expression, a state, and a new state.
  - Evaluation of a command has side effects but no direct result
  - The "result" of a \( \text{Comm} \) is a new state: \( \langle c, \sigma \rangle \Downarrow \sigma' \)
  - The evaluation of a command might not terminate.
Evaluation Rules (for Comm)

\[ \langle e, \sigma \rangle \uparrow n \]
\[ \langle x := e, \sigma \rangle \uparrow \sigma[x := n] \]
\[ \langle \text{skip}, \sigma \rangle \uparrow \sigma \]
\[ \langle c_1 \land c_2, \sigma \rangle \uparrow \sigma' \]
\[ \langle c_1, \sigma \rangle \uparrow \sigma' \]
\[ \langle b \land \text{true}, \sigma \rangle \uparrow \sigma' \]
\[ \langle b \land \text{false}, \sigma \rangle \uparrow \sigma' \]
\[ \langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma \rangle \uparrow \sigma' \]
\[ \langle \text{while } b \text{ do } c, \sigma \rangle \uparrow \sigma' \]

Def:
\[ \sigma[x := n](x) = n \]
\[ \sigma[x := n](y) = \sigma(y) \]

Evaluation of Commands: Notes

- The order of evaluation is important and explicit.
  - \( c_1 \) is evaluated before \( c_2 \) in \( c_1 ; c_2 \).
  - \( c_2 \) is not evaluated in \( "if true then c_1 else c_2" \).
  - \( c \) is not evaluated in \( "while false do c" \).
  - \( b \) is evaluated first in \( "if b then c_1 else c_2" \).
- The evaluation rules are not syntax-directed.
  - See the rule for \( \text{while} \).
- The evaluation rules do suggest an interpreter.
- Conditional constructs have multiple evaluation rules, but only one can be applied at one time.
- Evaluate from the zero-initialized store:
  while \( x < 5 \) do \( x := x + 1 \)

Homework 2 (for Tuesday 4 Oct 05)

- Write OCAML code to implement the operational semantics of IMP
- Some guidelines
  - write types for \( \text{AExp} \), \( \text{BExp} \), \( \text{Comm} \)
  - represent variables as strings
  - represent stores as a function from variables to integers
- Handy documentation:

Homework 2 Template

```ocaml
type variable = string;;
type store = (variable -> int);;
let modif (f:store) (y:variable) (v:int) =
  function (x:variable) ->
    if x = y then v else f(x);;

let aexp =
| Int of int
| Loc of variable
| ...

let bexp = ...

type comm = ...

let c = While(Leq(Loc "x",Int 5), Set("x",Sum(Loc "x",Int 1)));
let r = commeval c zero_store;;
r("z");;
r("x");;
```

Cheating

- All work you turn in must be your own
- If you don’t know if something is allowed, please ask
- Any cheating will result in failure of the course and other standard measures
- You are encouraged to discuss course material and assignments with others
- You are not allowed do homeworks with others
- You may use any conversations, texts, or other material, as long as you cite your sources