University of Massachusetts Lowell — Comp 3010: Organization of Programming Languages

Assignment 6

Name: **[[*TODO: Harishwar Reddy Erri*]]**
UML ID: **[[*TODO: 02148304*]]**
Collaborators: **[[*TODO: Manasvi Boineypally (02148325)*]]**

Make sure that the remaining pages of this assignment do not contain any identifying information.

1: Grammar for Evaluation Contexts

In operational semantics, evaluation contexts help specify the order in which expressions are evaluated. Evaluation contexts are typically defined as parts of an expression where evaluation can still occur.

Given the language syntax provided:

- Expressions (e) include: - Variables (x) - Numbers (n) - Addition ($e_1 + e_2$) - Multiplication ($e_1 \times e_2$) - Assignment ($x := e_1; e_2$)

Let's define evaluation contexts E to isolate where the next expression to evaluate should be placed.

Grammar for Evaluation Contexts E

The evaluation context grammar, E, is defined as follows:

(a) $E ::= [\cdot]$ — the `hole` representing the place where evaluation occurs.

(b) $E ::= E + e$ — addition where the left subexpression is an evaluation context.

(c) $E ::= v + E$ — addition where the right subexpression is an evaluation context and the left subexpression is fully evaluated to a value v.

(d) $E ::= E \times e$ — multiplication where the left subexpression is an evaluation context.

(e) $E ::= v \times E$ — multiplication where the right subexpression is an evaluation context, with the left side evaluated to a value v.

(f) $E ::= x := E; e$ — assignment context, where the expression being assigned is evaluated.

(g) $E ::= v := n; E$ — assignment context, where the right-hand side (expression after assignment) is an evaluation context, given that the left assignment has been evaluated.

This grammar specifies the order of operations by defining where the `hole` should appear as we evaluate each component of an expression.

Small-Step Operational Semantics Using Evaluation Contexts

Using these evaluation contexts, we can define a new set of small-step semantics. Each semantic rule specifies how to reduce an expression step-by-step. We'll use notation like $\langle E[e], \sigma\rangle \to \langle E[e'], \sigma'\rangle$ to denote that expression e within an evaluation context E reduces to e' with potentially updated state σ.

Rules for the Semantics

(a) **Variable Lookup (VAR)**

$$\frac{}{\langle x, \sigma\rangle \to \langle n, \sigma\rangle} \quad \text{where } n = \sigma(x)$$

This rule states that a variable x evaluates to its value n in the environment σ.

(b) **Addition (LADD and RADD)**

- **Left Addition (LADD)**

$$\frac{\langle e_1, \sigma\rangle \to \langle e_1', \sigma'\rangle}{\langle e_1 + e_2, \sigma\rangle \to \langle e_1' + e_2, \sigma'\rangle}$$

This rule applies if the left operand of an addition can be further reduced.

- **Right Addition (RADD)**

$$\frac{\langle e_2, \sigma\rangle \to \langle e_2', \sigma'\rangle}{\langle n + e_2, \sigma\rangle \to \langle n + e_2', \sigma'\rangle}$$

Here, the right operand is reduced if the left operand is already a value.

(c) **Addition with Values (ADD)**

$$\frac{}{\langle n + m, \sigma \rangle \to \langle p, \sigma \rangle} \quad \text{where } p = n + m$$

This rule reduces an addition of two values directly to their sum.

(d) **Multiplication (LMUL and RMUL)**

- **Left Multiplication (LMUL)**

$$\frac{\langle e_1, \sigma \rangle \to \langle e_1', \sigma' \rangle}{\langle e_1 \times e_2, \sigma \rangle \to \langle e_1' \times e_2, \sigma' \rangle}$$

  Similar to addition, this applies if the left operand can be reduced.

- **Right Multiplication (RMUL)**

$$\frac{\langle e_2, \sigma \rangle \to \langle e_2', \sigma' \rangle}{\langle n \times e_2, \sigma \rangle \to \langle n \times e_2', \sigma' \rangle}$$

  Here, the right operand is reduced if the left operand is a value.

(e) **Multiplication with Values (MUL)**

$$\frac{}{\langle n \times m, \sigma \rangle \to \langle p, \sigma \rangle} \quad \text{where } p = n \times m$$

This rule reduces a multiplication of two values directly to their product.

(f) **Assignment (ASG1 and ASG)**

- **Evaluate Assignment Expression (ASG1)**

$$\frac{\langle e_1, \sigma \rangle \to \langle e_1', \sigma' \rangle}{\langle x := e_1; e_2, \sigma \rangle \to \langle x := e_1'; e_2, \sigma' \rangle}$$

  The assignment expression e_1 is evaluated first before it is assigned.

- **Perform Assignment (ASG)**

$$\frac{}{\langle x := n; e_2, \sigma \rangle \to \langle e_2, \sigma[x \mapsto n] \rangle}$$

  Once the assignment expression is fully evaluated, x is assigned the value n in the environment σ, and then we proceed to evaluate e_2.

2: More Evlaution Contexts

Let's go through the evaluation of the expression step by step using small-step semantics with evaluation contexts. Each step will specify the evaluation context E and the expression e being evaluated.

The expression given is:

$$\text{let } y = (\lambda x.\, x, ((\lambda x.\, \lambda y.\, y)(\lambda z.\, z), \lambda x.\, x)) \text{ in } (\lambda z.\, \#2\, y)\, y$$

To make this more readable, let's break it down into components and evaluate each part according to the rules. We assume here that we have rules for evaluating tuples, function application, and lambda expressions.

Step 1: Desugaring the 'let' Expression

The 'let' expression 'let y = ⟨value⟩ in ⟨body⟩' is syntactic sugar for substitution, where we bind y to the expression on the right and then evaluate the body with this binding. We start by rewriting the expression:

$$(\lambda y.\, (\lambda z.\, \#2\, y)\, y)\, (\lambda x.\, x, ((\lambda x.\, \lambda y.\, y)(\lambda z.\, z), \lambda x.\, x))$$

Now, we proceed with a step-by-step evaluation.

Step 2: Applying the 'let' Binding

Substitute y with $(\lambda x.\, x, ((\lambda x.\, \lambda y.\, y)(\lambda z.\, z), \lambda x.\, x))$ in the body $(\lambda z.\, \#2\, y)\, y$:

$$(\lambda z.\, \#2\, (\lambda x.\, x, ((\lambda x.\, \lambda y.\, y)(\lambda z.\, z), \lambda x.\, x)))\, (\lambda x.\, x, ((\lambda x.\, \lambda y.\, y)(\lambda z.\, z), \lambda x.\, x))$$

Now we need to evaluate this function application by reducing it step by step.

Step 3: Evaluate the Outer Application

We have an application $(\lambda z.\, \#2\, y)\, y$ with y substituted. The expression $\#2\, y$ denotes the second element of the tuple assigned to y, so we proceed to evaluate this tuple access.

Step 4: Accessing the Second Element of the Tuple

The tuple is:

$$(\lambda x.\, x, ((\lambda x.\, \lambda y.\, y)(\lambda z.\, z), \lambda x.\, x))$$

The second element of this tuple is:

$$((\lambda x.\, \lambda y.\, y)(\lambda z.\, z), \lambda x.\, x)$$

Thus, $\#2\, y$ reduces to:

$$((\lambda x.\, \lambda y.\, y)(\lambda z.\, z), \lambda x.\, x)$$

Step 5: Substitute and Evaluate

Now, we substitute this result back into the expression:

$$(\lambda z.\, ((\lambda x.\, \lambda y.\, y)(\lambda z.\, z), \lambda x.\, x))\, (\lambda x.\, x, ((\lambda x.\, \lambda y.\, y)(\lambda z.\, z), \lambda x.\, x))$$

Here, we need to evaluate each component, starting with the innermost function applications.

Step 6: Evaluate $(\lambda x.\, \lambda y.\, y)(\lambda z.\, z)$

Applying $(\lambda x.\, \lambda y.\, y)$ to $(\lambda z.\, z)$:

(a) Substitute x with $\lambda z.\, z$ in $\lambda y.\, y$.

(b) This gives us $\lambda y.\, y$.

So, $(\lambda x.\, \lambda y.\, y)(\lambda z.\, z)$ reduces to $\lambda y.\, y$.

Step 7: Final Expression

Now we substitute back into the remaining expression:

$$(\lambda z.\, (\lambda y.\, y, \lambda x.\, x))\, (\lambda x.\, x, (\lambda y.\, y, \lambda x.\, x))$$

Here we evaluate by applying λz to the tuple, leading us to further reductions.

3: Translation Definition

To solve this, let's address each part of the question, breaking down the translation of the extended IMP language (with the 'unless' command) to vanilla IMP. The 'unless' command in this extended IMP language behaves as follows:

- c unless b executes the command c if the boolean expression b is **false**. - It is equivalent to if b then skip else c.

Part (a): Define the Translation from Extended IMP to Vanilla IMP

Goal:

Define a translation T that translates arithmetic expressions, boolean expressions, and commands from the extended IMP language with 'unless' into vanilla IMP, which does not include 'unless'.

Step-by-Step Translation Definition

Let's fill in the translation rules for the elements of the extended IMP language.

1. **Arithmetic Expressions**

- Translate a constant number n:
$$T[n] = n$$
- Translate addition $a_1 + a_2$:
$$T[a_1 + a_2] = T[a_1] + T[a_2]$$
- Translate multiplication $a_1 \times a_2$:
$$T[a_1 \times a_2] = T[a_1] \times T[a_2]$$

2. **Boolean Expressions**

- For boolean expressions, we assume translations like:
$$T[\text{true}] = \text{true} \quad \text{and} \quad T[\text{false}] = \text{false}$$
- Translate equality $a_1 = a_2$:
$$T[a_1 = a_2] = T[a_1] = T[a_2]$$
- Translate less-than comparison $a_1 \langle a_2$:
$$T[a_1 \langle a_2] = T[a_1] \langle T[a_2]$$

3. **Commands**

- For the standard commands in IMP, the translation is straightforward:
  - **Skip**: $T[\text{skip}] = \text{skip}$
  - **Assignment**: For an assignment $x := a$,
$$T[x := a] = x := T[a]$$
  - **Sequence**: For a sequence of commands $c_1; c_2$,
$$T[c_1; c_2] = T[c_1]; T[c_2]$$
  - **Conditional**: For an 'if' statement if b then c_1 else c_2,
$$T[\text{if } b \text{ then } c_1 \text{ else } c_2] = \text{if } T[b] \text{ then } T[c_1] \text{ else } T[c_2]$$

  - **While Loop**: For a 'while' loop while b do c,

$$T[\text{while } b \text{ do } c] = \text{while } T[b] \text{ do } T[c]$$

- **Unless**: For the 'unless' command c unless b,

$$T[c \text{ unless } b] = \text{if } T[b] \text{ then skip else } T[c]$$

  This translation replaces c unless b with the 'if-then-else' construct in vanilla IMP, where c is executed only if b is false.

Part (b): Is the Translation Sound?

To determine if the translation is sound, we refer to the concept of **soundness of translation adequacy**. A translation is sound if every behavior (execution result) of the translated program in the target language (vanilla IMP) corresponds to a behavior in the source language (extended IMP).

In this case: - The translation of c unless b to if b then skip else c correctly mirrors the intended behavior of 'unless' in the source language. - Each expression and command in extended IMP has a corresponding behavior in vanilla IMP through the translation function T.

Justification: Since the behavior of 'unless' in extended IMP matches the if b then skip else c construct in vanilla IMP, the translation does not introduce any new behaviors that wouldn't exist in the extended language. Therefore, **the translation is sound**.

Part (c): Is the Translation Complete?

To determine if the translation is complete, we refer to the **completeness of translation adequacy**. A translation is complete if every behavior in the source language (extended IMP) can be reproduced in the target language (vanilla IMP) after translation.

In this case: - For each arithmetic expression, boolean expression, and command in the extended IMP language, there exists a corresponding construct or sequence of constructs in vanilla IMP that behaves identically after translation. - The translation captures the exact behavior of c unless b with if b then skip else c, ensuring that there is no loss of behavior from extended IMP to vanilla IMP.

Justification: Since every behavior in the extended IMP language has an equivalent in vanilla IMP through the translation T, **the translation is complete**.