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

Assignment 6

Name: Ashish Kosana

UML ID: 02148256

Collaborators: NONE

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

1 Evaluation Contexts (20 points)

Here is the syntax of a variant of an arithmetic expression language with assignment. The small-step semantics are also given below. (Time to refresh: you have already seen this language in the midterm exam)

$$e ::= x \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid x := e_1; e_2$$

$$v ::= n$$

Small-step operational semantics

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

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

Give a grammar for evaluation contexts as well as given the new set of small-step semantics using evaluation contexts.

Answer:

Grammar for evaluation contexts:

$$E ::= [] \mid E + e \mid n + E \mid E \times e \mid n \times E \mid x := E; e$$

Small-step semantics using evaluation contexts:

$$\text{CTXT}\ \frac{\langle e, \sigma\rangle \to \langle e', \sigma'\rangle}{\langle E[e], \sigma\rangle \to \langle E[e'], \sigma'\rangle}$$

$$\text{VAR}\ \frac{}{\langle E[x], \sigma\rangle \to \langle E[n], \sigma\rangle}\ \text{where } n = \sigma(x)$$

$$\text{ADD}\ \frac{}{\langle E[n + m], \sigma\rangle \to \langle E[p], \sigma\rangle}\ \text{where } p \text{ is the sum of } n \text{ and } m$$

$$\text{MUL}\ \frac{}{\langle E[n \times m], \sigma\rangle \to \langle E[p], \sigma\rangle}\ \text{where } p \text{ is the product of } n \text{ and } m$$

$$\text{ASG}\ \frac{}{\langle E[x := n; e], \sigma\rangle \to \langle E[e], \sigma[x \mapsto n]\rangle}$$

Detailed Explanation:

The evaluation context grammar defines the structure of expressions with a `hole` (represented by []) where the next reduction step can occur. This allows us to focus on the specific part of the expression that is being evaluated.

1. [] represents the hole where evaluation can occur. 2. $E + e$ and $n + E$ allow evaluation on either side of an addition operation. 3. $E \times e$ and $n \times E$ allow evaluation on either side of a multiplication operation. 4. $x := E; e$ allows evaluation in the right-hand side of an assignment before the assignment takes place.

The new set of small-step semantics using evaluation contexts simplifies the rules by incorporating the context E:

1. CTXT (Context): This is the key rule that allows reduction to occur in any evaluation context. It says that if an expression e can take a step to e' in some state σ, then e in any context E can also take that step.

2. VAR (Variable): This rule looks up the value of a variable in the current state, replacing the variable with its value.

3. ADD (Addition): This rule performs addition when both operands are numbers.

4. MUL (Multiplication): This rule performs multiplication when both operands are numbers.

5. ASG (Assignment): This rule updates the state with a new value for the assigned variable and continues with the rest of the expression.

This approach allows for a more concise set of rules while maintaining the same semantics as the original small-step rules. It also makes it easier to reason about the order of evaluation in complex expressions, as the evaluation context clearly shows which part of the expression is currently being evaluated.

2 More Evaluation Contexts (10 points)

Evaluate the following expression until it is a value using the small step rules given in Lecture 6. Assume integers are supported and are treated as values. Your solution must show all single steps (not just the final answer); each time you apply the CTXT rule, you should specify what your E and e is.

let y = (x. x, ((x. y. y) (z. z), x. x)) in (z.2 y) y

Answer:

Initial Expression: let $y = (\lambda x.\,.x, ((\lambda x.\,.\lambda y.\,.y)(\lambda z.\,.z), \lambda x.\,.x))$ in $(\lambda z.\,.\#2y)y$

$\rightarrow (\lambda z.\,.\#2(\lambda x.\,.x, ((\lambda x.\,.\lambda y.\,.y)(\lambda z.\,.z), \lambda x.\,.x)))(\lambda x.\,.x, ((\lambda x.\,.\lambda y.\,.y)(\lambda z.\,.z), \lambda x.\,.x))$
where $E = []$ and
$e =$ let $y = (\lambda x.\,.x, ((\lambda x.\,.\lambda y.\,.y)(\lambda z.\,.z), \lambda x.\,.x))$ in $(\lambda z.\,.\#2y)y$

$\rightarrow \#2(\lambda x.\,.x, ((\lambda x.\,.\lambda y.\,.y)(\lambda z.\,.z), \lambda x.\,.x))$
where $E = []$ and
$e = (\lambda z.\,.\#2(\lambda x.\,.x, ((\lambda x.\,.\lambda y.\,.y)(\lambda z.\,.z), \lambda x.\,.x)))(\lambda x.\,.x, ((\lambda x.\,.\lambda y.\,.y)(\lambda z.\,.z), \lambda x.\,.x))$

$\rightarrow ((\lambda x.\,.\lambda y.\,.y)(\lambda z.\,.z), \lambda x.\,.x)$
where $E = []$ and
$e = \#2(\lambda x.\,.x, ((\lambda x.\,.\lambda y.\,.y)(\lambda z.\,.z), \lambda x.\,.x))$

$\rightarrow (\lambda y.\,.y, \lambda x.\,.x)$
where $E = ([], \lambda x.\,.x)$ and
$e = (\lambda x.\,.\lambda y.\,.y)(\lambda z.\,.z)$

$\rightarrow \lambda x.\,.x$
where $E = []$ and $e = (\lambda y.\,.y, \lambda x.\,.x)$

Detailed Explanation: 1. We start with the let expression and apply the let-reduction rule, substituting y in the body. 2. We then apply the function application rule, substituting the argument into the body of the lambda. 3. The 2 operator selects the second element of the pair. 4. We evaluate the first element of the inner pair, applying (x. y. y) to (z. z). 5. Finally, we select the second element of the resulting pair, which is (x. x).

Each step shows the evaluation context E and the sub-expression e being evaluated, demonstrating how the CTXT rule is applied at each step.

3 Translation Definition (20 points)

Suppose we extend the IMP language with the command c unless b. Intuitively, this command executes command c if (and only if) boolean expression b evaluates to false. It is equivalent to the command if b then skip else c.

(a) Define the translation definition from IMP extended with unless to vanilla IMP. Refer to Lecture 4 for the vanilla IMP language. I'll show you the first few steps and you have to fill the remaining ones for arithmetic and boolean expressions as well as commands. Note the input language has unless command but the output language does not have one.

$$\begin{array}{lcl} \mathcal{T}[\![n]\!] & = & n \\ \mathcal{T}[\![a_1 + a_2]\!] & = & a_1 + a_2 \\ \mathcal{T}[\![a_1 \times a_2]\!] & = & \textbf{??} \\ \ldots & & \end{array}$$

Answer:

$$
\begin{array}{lcl}
\mathcal{T}[\![n]\!] & = & n \\
\mathcal{T}[\![a_1 + a_2]\!] & = & \mathcal{T}[\![a_1]\!] + \mathcal{T}[\![a_2]\!] \\
\mathcal{T}[\![a_1 \times a_2]\!] & = & \mathcal{T}[\![a_1]\!] \times \mathcal{T}[\![a_2]\!] \\
\mathcal{T}[\![x]\!] & = & x \\
\mathcal{T}[\![true]\!] & = & true \\
\mathcal{T}[\![false]\!] & = & false \\
\mathcal{T}[\![a_1 = a_2]\!] & = & \mathcal{T}[\![a_1]\!] = \mathcal{T}[\![a_2]\!] \\
\mathcal{T}[\![a_1 \leq a_2]\!] & = & \mathcal{T}[\![a_1]\!] \leq \mathcal{T}[\![a_2]\!] \\
\mathcal{T}[\![\neg b]\!] & = & \neg \mathcal{T}[\![b]\!] \\
\mathcal{T}[\![b_1 \wedge b_2]\!] & = & \mathcal{T}[\![b_1]\!] \wedge \mathcal{T}[\![b_2]\!] \\
\mathcal{T}[\![skip]\!] & = & skip \\
\mathcal{T}[\![x := a]\!] & = & x := \mathcal{T}[\![a]\!] \\
\mathcal{T}[\![c_1; c_2]\!] & = & \mathcal{T}[\![c_1]\!]; \mathcal{T}[\![c_2]\!] \\
\mathcal{T}[\![if b then c_1 else c_2]\!] & = & if \mathcal{T}[\![b]\!] then \mathcal{T}[\![c_1]\!] else \mathcal{T}[\![c_2]\!] \\
\mathcal{T}[\![while b do c]\!] & = & while \mathcal{T}[\![b]\!] do \mathcal{T}[\![c]\!] \\
\mathcal{T}[\![c \text{ unless } b]\!] & = & if \mathcal{T}[\![b]\!] then skip else \mathcal{T}[\![c]\!]
\end{array}
$$

Explanation: This translation definition maps each construct of the extended IMP language to its equivalent in vanilla IMP. Most constructs are translated directly, with the translation applied recursively to sub-expressions. The key addition is the translation of the 'unless' construct, which is mapped to an equivalent 'if-then-else' statement in vanilla IMP.

(b) Is your translation sound? Justify. (Refer to the Lecture 6 for the definition of Soundness of Translation Adequacy.)

Answer: Yes, this translation is sound.

Soundness of translation adequacy requires that for every program P in the source language (extended IMP), if P evaluates to some result R, then the translated program T(P) in the target language (vanilla IMP) also evaluates to R.

Justification: 1. For all constructs except 'unless', the translation is identity or a direct mapping that preserves semantics. 2. For the 'unless' construct, the translation $\mathcal{T}[\![c unless b]\!]$ = if $\mathcal{T}[\![b]\!]$ then skip else $\mathcal{T}[\![c]\!]$ preserves the original semantics: - If b is true, 'c unless b' does nothing, which is equivalent to 'skip' in the translation. - If b is false, 'c unless b' executes c, which is what the 'else' branch does in the translation. 3. The translation is applied recursively to all sub-expressions and sub-commands, ensuring that the semantics are preserved at all levels of the program structure.

Therefore, the behavior of any program in the extended IMP language will be preserved when translated to vanilla IMP, satisfying the soundness criterion.

(c) Is your translation complete? Justify. (Refer to the Lecture 6 for the definition of Completeness of Translation Adequacy.)

Answer: Yes, this translation is complete.

Completeness of translation adequacy requires that for every program P' in the target language (vanilla IMP), there exists a program P in the source language (extended IMP) such that T(P) = P'.

Justification: 1. The translation function T is defined for all constructs in the extended IMP language, including the new 'unless' command. 2. For every construct in vanilla IMP, there is a corresponding construct or combination of constructs in extended IMP that, when translated, produces that vanilla IMP construct: - All basic constructs (variables, constants, arithmetic operations, boolean operations, assignments, if-then-else, while) exist in both languages and are translated directly. - The 'unless' construct in extended IMP, when translated, produces an if-then-else construct in vanilla IMP. However, this does not prevent completeness, as any if-then-else in vanilla IMP can be represented directly in extended IMP. 3. Any program in vanilla IMP can be represented by an identical program in extended IMP (since extended IMP is a superset of vanilla IMP), which will then translate back to the original vanilla IMP program.

Therefore, for any program in vanilla IMP, we can construct an equivalent program in extended IMP that, when translated, yields the original vanilla IMP program. This satisfies the completeness criterion.