IMP Syntax and Semantics

1. Language Characteristics

These are the language features of IMP:

- Dynamic (weak) typing
- Core constructs: sequencing, assignment, while loops, if statements, arithmetic, string, and boolean operations, and input/output

2. Abstract Syntax

This section defines the abstract syntax of a IMP program. Note that this is not the same as the *concrete* syntax (i.e., what the programmer types in); this syntax describes the form of the abstract syntax tree generated by the parser.

Meta-variables for integers, booleans, strings, and variables:

$$n \in \mathbb{Z} \qquad b \in \mathit{Bool} \qquad \mathit{str} \in \mathit{String} \qquad x \in \mathit{Variable}$$

A program is a set of function definitions followed by a term, where a term is either a command or an expression.

$$\begin{aligned}
p \in \mathit{Program} &::= t \\
t \in \mathit{Term} &::= c \mid e \\
c \in \mathit{Cmd} &::= t_1 \,;\, t_2 \mid x := e \mid \textbf{while}\ e\ t \mid \textbf{output}\ e \\
e \in \mathit{Exp} &::= n \mid b \mid \mathit{str} \mid \textbf{unit} \mid x \mid \neg e \\
&\quad \mid e_1 \oplus e_2 \mid \textbf{if}\ e\ t_1\ \textbf{else}\ t_2 \mid \textbf{input}\ \mathit{typ} \\
&\quad \mid \textbf{var}\ \overrightarrow{x = e}\ \textbf{in}\ t \\
\mathit{typ} \in \mathit{InputType} &::= \textbf{num} \mid \textbf{str} \\
\oplus \in \mathit{BinOp} &::= + \mid - \mid \times \mid \div \mid \wedge \mid \vee \mid = \mid \leq
\end{aligned}$$

3. Semantic Domains

This section describes the mathematical objects that we will use to describe the semantics of IMP. You can think of each domain (i.e., *Config*, *Store*, etc) as a *type*, and these definitions are describing what kind of thing each type represents.

The abstract machine configuration:

$$C \in \mathit{Config} = \mathit{Term} \times \mathit{Store}$$

The configuration subterms:

$$\sigma \in \mathit{Store} = \mathit{Variable} \to \mathit{Value}$$

The language values:

$$v \in \mathit{Value} = \mathbb{Z} + \mathit{Bool} + \mathit{String} + \textbf{unit}$$

4. Semantic Helper Functions

These helper functions are used to help evaluate a IMP program.

The `inject` helper function takes a program and returns the initial configuration of the abstract machine:

$$\texttt{inject} \in \mathit{Program} \to \mathit{Config}$$

$$\texttt{inject}(t) = t \cdot \sigma \quad \text{where} \quad \sigma = \emptyset$$

5. IMP Semantics

These are the semantic rules for evaluating IMP programs. Each rule takes a specific kind of configuration and specifies how to resolve that kind of configuration to a (*Value*, *Store*) pair. Together they define the evaluation function:

$$[\![\cdot]\!] \in \mathit{Config} \to \mathit{Value} \times \mathit{Store}$$

The sequencing command is evaluated by first evaluating the left-hand side term, then discarding the result and evaluating the right-hand side term.

$$\frac{[\![t_1 \cdot \sigma]\!] \Rightarrow (v, \sigma_1)}{[\![t_1 \,;\, t_2 \cdot \sigma]\!] \Rightarrow [\![t_2 \cdot \sigma_1]\!]} \quad (\textsc{seq})$$

The assignment command is evaluated by first evaluating the right-hand side expression, then modifying the store to map the variable x to the value of that expression. If x is not already in the store then the behavior is undefined.

$$\frac{[\![e \cdot \sigma]\!] \Rightarrow (v, \sigma_1) \qquad x \in \mathit{dom}(\sigma_1) \qquad \sigma_2 = \sigma_1[x \mapsto v]}{[\![x := e \cdot \sigma]\!] \Rightarrow (\textbf{unit}, \sigma_2)} \quad (\textsc{asn})$$

The while command is evaluated in one of two ways depending on the value of the guard expression. If the guard expression evaluates to **true** then the body of the command is evaluated and then the same while command is evaluated over again (with the new store).

$$\frac{[\![e \cdot \sigma]\!] \Rightarrow (\textbf{true}, \sigma_1) \qquad [\![t \cdot \sigma_1]\!] \Rightarrow (v, \sigma_2)}{[\![\textbf{while}\ e\ t \cdot \sigma]\!] \Rightarrow [\![\textbf{while}\ e\ t \cdot \sigma_2]\!]} \quad (\textsc{while-t})$$

If the guard expression evaluates to **false** then the while expression evaluates to **unit**, terminating the loop. If the guard expression does not evaluate to a boolean then the behavior is undefined.

$$\frac{[\![e \cdot \sigma]\!] \Rightarrow (\textbf{false}, \sigma_1)}{[\![\textbf{while}\ e\ t \cdot \sigma]\!] \Rightarrow (\textbf{unit}, \sigma_1)} \quad (\textsc{while-f})$$

The output command is evaluated by first evaluating the given expression, then transmitting the resulting value.

$$\frac{[\![e \cdot \sigma]\!] \Rightarrow (v, \sigma_1)}{[\![\textbf{output}\ e \cdot \sigma]\!] \overset{v!}{\Rightarrow} (\textbf{unit}, \sigma_1)} \quad (\textsc{out})$$

A value $v \in \mathit{Value}$ always evaluates to itself. In other words, a syntactic number, boolean, string, or **unit** will evaluate into its semantic equivalent.

$$[\![v \cdot \sigma]\!] \Rightarrow (v, \sigma) \qquad \text{(VAL)}$$

A variable is evaluated by looking it up in the store to get the associated value. If x isn't in the store then the behavior is undefined.

$$\frac{\sigma(x) = v}{[\![x \cdot \sigma]\!] \Rightarrow (v, \sigma)} \qquad \text{(VAR)}$$

Logical negation is evaluated by first evaluating the negated expression to a Boolean value, then negating that value. If the expression doesn't evaluate to a Boolean then the behavior is undefined.

$$\frac{[\![e \cdot \sigma]\!] \Rightarrow (b, \sigma_1)}{[\![\neg e \cdot \sigma]\!] \Rightarrow (\neg b, \sigma_1)} \qquad \text{(NOT)}$$

The Boolean binary operators $\wedge, \vee$ are evaluated by evaluating the left-hand side and right-hand side expressions to Boolean values, then applying $\wedge$ or $\vee$ to the result. If either expression does not evaluate to a Boolean then the behavior is undefined.

$$\frac{[\![e_1 \cdot \sigma]\!] \Rightarrow (b_1, \sigma_1) \qquad [\![e_2 \cdot \sigma_1]\!] \Rightarrow (b_2, \sigma_2) \qquad \oplus \in \{\wedge, \vee\}}{[\![e_1 \oplus e_2 \cdot \sigma]\!] \Rightarrow (b_1 \oplus b_2, \sigma_2)} \qquad \text{(BOP)}$$

The Arithmetic binary operators $+, -, \times, \div, \leq$ are evaluated by evaluating the left-hand side and right-hand side expressions to integer values, then applying the appropriate operator to the results. If the expressions evaluate to strings instead of integers see the rule SOP; otherwise if either expression does not evaluate to an integer or the operator is $\div$ and the right-hand side expression evaluates to 0, then the behavior is undefined.

$$\frac{[\![e_1 \cdot \sigma]\!] \Rightarrow (n_1, \sigma_1) \qquad [\![e_2 \cdot \sigma_1]\!] \Rightarrow (n_2, \sigma_2) \qquad \oplus \in \{+, -, \times, \div, \leq\} \qquad n_2 = 0 \supset \oplus \neq \div}{[\![e_1 \oplus e_2 \cdot \sigma]\!] \Rightarrow (n_1 \oplus n_2, \sigma_2)} \qquad \text{(AOP)}$$

The String binary operators $+, \leq$ are evaluated by evaluating the left-hand side and right-hand side expressions to string values, then applying the appropriate operator to the results (where $+$ is concatenation and $\leq$ is lexicographic comparison). If the expressions evaluate to integers instead of strings see the rule AOP; otherwise if either expression does not evaluate to an string then the behavior is undefined.

$$\frac{[\![e_1 \cdot \sigma]\!] \Rightarrow (str_1, \sigma_1) \qquad [\![e_2 \cdot \sigma_1]\!] \Rightarrow (str_2, \sigma_2) \qquad \oplus \in \{+, \leq\}}{[\![e_1 \oplus e_2 \cdot \sigma]\!] \Rightarrow (str_1 \oplus str_2, \sigma_2)} \qquad \text{(SOP)}$$

The Equality binary operator is treated separately from the others because the previous rules all require that the two operands evaluate to the same type. For equality we want to be able to compare values to each other regardless of types.

$$\frac{[\![e_1 \cdot \sigma]\!] \Rightarrow (v_1, \sigma_1) \qquad [\![e_2 \cdot \sigma_1]\!] \Rightarrow (v_2, \sigma_2)}{[\![e_1 = e_2 \cdot \sigma]\!] \Rightarrow (v_1 \stackrel{?}{=} v_2, \sigma_2)} \qquad \text{(EQ)}$$

The if expression is also evaluated in one of two ways depending on the value of the guard expression. If the guard expression evaluates to **true** then the term in the true branch is evaluated.

$$\frac{[\![e \cdot \sigma]\!] \Rightarrow (\textbf{true}, \sigma_1)}{[\![\textbf{if } e \ t_1 \ \textbf{else } t_2 \cdot \sigma]\!] \Rightarrow [\![t_1 \cdot \sigma_1]\!]} \qquad \text{(IF-T)}$$

If the guard expression evaluates to **false** then the term in the false branch is evaluated. If the guard expression does not evaluate to a boolean then the behavior is undefined.

$$\frac{[\![e \cdot \sigma]\!] \Rightarrow (\textbf{false}, \sigma_1)}{[\![\textbf{if } e \ t_1 \ \textbf{else } t_2 \cdot \sigma]\!] \Rightarrow [\![t_2 \cdot \sigma_1]\!]} \qquad \text{(IF-F)}$$

The input expression is evaluated by receiving and returning a value. If the received value is not either an integer or a string then the behavior is undefined.

$$[\![\textbf{input } typ \cdot \sigma]\!] \stackrel{v?}{\Rightarrow} (v, \sigma) \qquad \text{(IN)}$$

A block of code is evaluated by first evaluating the variable binding expressions and binding the resulting values to the respective variables in the store, then evaluating the block's term.

$$\frac{[\![\vec{e} \cdot \sigma]\!] \Rightarrow (\vec{v}, \sigma_1) \qquad \sigma_2 = \sigma_1[\overrightarrow{x \mapsto v}]}{[\![\textbf{var } \overrightarrow{x = e} \textbf{ in } t \cdot \sigma]\!] \Rightarrow [\![t \cdot \sigma_2]\!]} \qquad \text{(BLOCK)}$$