Operational Semantics of a Simple Imperative Language*

– lecture notes –

Formal methods provide instruments for defining and reasoning about programming languages. A key aspect of this is specifying the *semantics* of a language—the precise meaning of its constructs. There are three main approaches to semantics:

- **Operational semantics**: Defines meaning through execution steps or evaluation rules
- **Denotational semantics**: Maps programs to mathematical objects in a semantic domain
- **Axiomatic semantics**: Program behavior is captured through logical assertions and proof rules

In these notes, we focus on operational semantics, specifically the big-step (or natural) semantics style. Axiomatic semantics, including Hoare Logic, will be addressed in a different document.

1 The IMP Language

To illustrate these concepts, we introduce **IMP**, a minimal imperative programming language. Despite its simplicity, IMP captures the essential features of imperative programming: arithmetic and boolean expressions, variable assignment, sequential composition, conditional branching, and loops.

The syntax of IMP is defined by the following BNF grammar:

$$
\begin{aligned}
&n \in \mathbb{Z} \\
&x \in \mathrm{Id} \\
&a ::= n \mid x \mid a_1 + a_2 \mid a_1 \times a_2 \\
&b ::= \texttt{true} \mid \texttt{false} \mid a_1 < a_2 \mid \neg b \mid b_1 \wedge b_2 \\
&S ::= \texttt{skip} \mid x := a \mid S_1; S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \mid \texttt{while } b \texttt{ do } S
\end{aligned}
$$

where a ranges over arithmetic expressions, b over boolean expressions, and S over statements. IMP is sufficient for our purposes because:

- It includes the fundamental control flow constructs (sequencing, conditionals, loops)
- It is simple enough to allow clear, manageable formal definitions and proofs
- It is expressive enough to write non-trivial programs
- The techniques developed for IMP extend naturally to more complex languages

*The content of this document is not claimed to be original or to be published as original research. It is meant to serve as a learning resource for students.

1.1 Syntax of IMP in Dafny

We begin with the syntax of our imperative language, consisting of identifiers, arithmetic expressions, boolean expressions, and statements. There is a natural encoding of the BNF grammar of IMP in Dafny.

1.1.1 Identifiers

```
datatype Id = x | y | z | s | t | u | v | n | m | i | j | g
```

1.2 Arithmetic Expressions

```
datatype AExp =
    Num(int)
  | Var(Id)
  | Plus(AExp, AExp)
  | Times(AExp, AExp)
```

1.3 Boolean Expressions

```
datatype BExp =
  | B(bool)
  | Less(AExp, AExp)
  | Not(BExp)
  | And(BExp, BExp)
```

1.4 Statements

```
datatype Stmt =
    Skip
  | Assign(Id, AExp)
  | Seq(Stmt, Stmt)
  | If(BExp, Stmt, Stmt)
  | While(BExp, Stmt)
```

2 Semantics

We define the meaning of each language construct using big-step operational semantics. The first most important part is to define what is the state of an IMP program. Naturally, the state is just a partial function from variables to values:

$$\sigma : \texttt{Id} \rightharpoonup \mathbb{Z}$$

The second step is to define what is a configuration, that is, all the needed information needed to describe what a program is doing at a particular point in time. For IMP, the configuration is a pair

$$\langle \star, \sigma \rangle,$$

where σ is the program state and $\star \in \{\mathit{Stmt}, \mathit{AExp}, \mathit{BExp}, \mathit{Id}, \mathbb{Z}\}$. The configuration can also have the form

$$\langle * \rangle,$$

where $* \in \mathbb{Z}$, since in that case, the second component (i.e., σ) of the pair does not play an important role.

Examples of Configurations.

Here are some concrete examples of configurations:

- $\langle x + 2, \sigma \rangle$ where $\sigma = \{x \mapsto 5\}$ — an arithmetic expression configuration
- $\langle 0 < x, \sigma \rangle$ where $\sigma = \{x \mapsto 3, y \mapsto 7\}$ — a boolean expression configuration
- $\langle x := y + 1, \sigma \rangle$ where $\sigma = \{y \mapsto 10\}$ — a statement configuration (assignment)
- $\langle \texttt{while}\ (0 < x)\ \texttt{do}\ x := x + (-1), \sigma \rangle$ where $\sigma = \{x \mapsto 2\}$ — a statement configuration (loop)
- $\langle 7 \rangle$ — a final value configuration (result of evaluating an arithmetic expression)
- $\langle \texttt{skip}, \sigma \rangle$ where $\sigma = \{x \mapsto 0, y \mapsto 1\}$ — the trivial statement with some state

2.1 States and Configurations in Dafny

```
type State = map<Id, int>
type Configuration = (Stmt, State)
```

2.2 Big-Step Semantics of Arithmetic Expressions

The evaluation of arithmetic expressions is defined by the following inference rules:

$$\textsc{Const}\ \frac{\cdot}{\langle n, \sigma \rangle \Downarrow \langle n \rangle}$$

$$\textsc{Lookup}\ \frac{\cdot}{\langle x, \sigma \rangle \Downarrow \sigma(x)}\ x \in \sigma$$

$$\textsc{Add}\ \frac{\langle a_1, \sigma \rangle \Downarrow n_1 \qquad \langle a_2, \sigma \rangle \Downarrow n_2}{\langle a_1 + a_2, \sigma \rangle \Downarrow \langle n_1 + n_2 \rangle}$$

$$\textsc{Mul}\ \frac{\langle a_1, \sigma \rangle \Downarrow n_1 \qquad \langle a_2, \sigma \rangle \Downarrow n_2}{\langle a_1 \times a_2, \sigma \rangle \Downarrow \langle n_1 \times n_2 \rangle}$$

Implementation in Dafny.

```
function evalAExp(a: AExp, s: State): int {
  match a {
    case Num(n) => n
    case Var(someVar) => if someVar in s then s[someVar] else 0
    case Plus(a1, a2) => evalAExp(a1, s) + evalAExp(a2, s)
    case Times(a1, a2) => evalAExp(a1, s) * evalAExp(a2, s)
  }
}
```

Example Derivation. Consider the arithmetic expression $x + 2$ and a state σ where $\sigma(x) = 2$. We show that $\langle x + 2, \sigma\rangle \Downarrow \langle 4\rangle$:

$$\textsc{Add}\ \frac{\textsc{Lookup}\ \dfrac{\cdot}{\langle x, \sigma\rangle \Downarrow 2}\ x \in \sigma \qquad \textsc{Const}\ \dfrac{\cdot}{\langle 2, \sigma\rangle \Downarrow \langle 2\rangle}}{\langle x + 2, \sigma\rangle \Downarrow \langle 4\rangle}$$

2.2.1 Big-Step Semantics of Boolean Expressions

The evaluation of boolean expressions is defined by the following inference rules:

$$\textsc{BTrue}\ \frac{\cdot}{\langle \texttt{true}, \sigma\rangle \Downarrow \langle \texttt{true}\rangle}$$

$$\textsc{BFalse}\ \frac{\cdot}{\langle \texttt{false}, \sigma\rangle \Downarrow \langle \texttt{false}\rangle}$$

$$\textsc{Less}\ \frac{\langle a_1, \sigma\rangle \Downarrow \langle n_1\rangle \qquad \langle a_2, \sigma\rangle \Downarrow \langle n_2\rangle}{\langle a_1 < a_2, \sigma\rangle \Downarrow (n_1 <_{\mathbb{Z}} n_2)}$$

$$\textsc{Not}\ \frac{\langle b, \sigma\rangle \Downarrow \langle v\rangle}{\langle \neg b, \sigma\rangle \Downarrow \langle !v\rangle}$$

$$\textsc{And}\ \frac{\langle b_1, \sigma\rangle \Downarrow \langle v_1\rangle \qquad \langle b_2, \sigma\rangle \Downarrow \langle v_2\rangle}{\langle b_1 \wedge b_2, \sigma\rangle \Downarrow \langle v_1\ \texttt{\&\&}\ v_2\rangle}$$

2.2.2 Implementation in Dafny.

```
28 function evalBExp(b: BExp, s: State): bool {
29   match b {
30     case B(bval) => bval
31     case Less(a1, a2) => evalAExp(a1, s) < evalAExp(a2, s)
32     case Not(b1) => !evalBExp(b1, s)
33     case And(b1, b2) => evalBExp(b1, s) && evalBExp(b2, s)
34   }
35 }
```

2.2.3 Example Derivation.

Consider the expression $0 < x$ where $\sigma(x) = 2$. We show that $\langle 0 < x, \sigma\rangle \Downarrow \texttt{true}$:

$$\textsc{Less}\ \frac{\textsc{Const}\ \dfrac{\cdot}{\langle 0, \sigma\rangle \Downarrow \langle 0\rangle} \qquad \textsc{Lookup}\ \dfrac{\cdot}{\langle x, \sigma\rangle \Downarrow \langle 2\rangle}\ x \in \sigma}{\langle 0 < x, \sigma\rangle \Downarrow \langle \texttt{true}\rangle}$$

2.2.4 Big-Step Semantics of Statements

The execution of statements is defined by the following inference rules:

$$\textsc{Skip}\ \frac{\cdot}{\langle \texttt{skip}, \sigma\rangle \Downarrow \langle \sigma\rangle}$$

$$\textsc{Assign}\ \frac{\langle a, \sigma\rangle \Downarrow \langle n\rangle}{\langle x := a, \sigma\rangle \Downarrow \langle \sigma[x \mapsto n]\rangle}$$

$$\textsc{Seq}\ \frac{\langle S_1, \sigma\rangle \Downarrow \langle \sigma''\rangle \qquad \langle S_2, \sigma''\rangle \Downarrow \langle \sigma'\rangle}{\langle S_1; S_2, \sigma\rangle \Downarrow \langle \sigma'\rangle}$$

$$\textsc{If-True}\ \frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle \qquad \langle S_1, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma\rangle \Downarrow \langle \sigma'\rangle}$$

$$\textsc{If-False}\ \frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle \qquad \langle S_2, \sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma\rangle \Downarrow \langle \sigma'\rangle}$$

$$\textsc{While-False}\ \frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{false}\rangle}{\langle \texttt{while } b \texttt{ do } S, \sigma\rangle \Downarrow \langle \sigma\rangle}$$

$$\textsc{While-True}\ \frac{\langle b, \sigma\rangle \Downarrow \langle \texttt{true}\rangle \qquad \langle S, \sigma\rangle \Downarrow \langle \sigma''\rangle \qquad \langle \texttt{while } b \texttt{ do } S, \sigma''\rangle \Downarrow \langle \sigma'\rangle}{\langle \texttt{while } b \texttt{ do } S, \sigma\rangle \Downarrow \langle \sigma'\rangle}$$

Example: While Loop Derivation.

Consider the program $\texttt{while}\ (0 < x)\ \texttt{do}\ x := x + (-1)$ where $\sigma_0(x) = 2$. We show that execution terminates in state σ_2 where $\sigma_2(x) = 0$.

We use the next notations:

- $W = \texttt{while}\ (0 < x)\ \texttt{do}\ x := x + (-1)$;
- $S = x := x + (-1)$;
- $\sigma_0 = \{x \mapsto 2\}$;
- $\sigma_1 = \{x \mapsto 1\}$;
- $\sigma_2 = \{x \mapsto 0\}$.

First iteration ($\sigma_0 \to \sigma_1$): the loop executes for the first time, the value of x is decreased by 1. The condition of the loop:

$$\textsc{Less}\ \frac{\textsc{Const}\ \dfrac{\cdot}{\langle 0, \sigma_0\rangle \Downarrow \langle 0\rangle} \qquad \textsc{Lookup}\ \dfrac{\cdot}{\langle x, \sigma_0\rangle \Downarrow \langle 2\rangle}\ x \in \sigma_0}{\langle 0 < x, \sigma_0\rangle \Downarrow \langle \texttt{true}\rangle}$$

The first execution of the body of the loop:

$$\textsc{Add}\ \frac{\textsc{Lookup}\ \dfrac{\cdot}{\langle x, \sigma_0\rangle \Downarrow \langle 2\rangle}\ x \in \sigma_0 \qquad \textsc{Const}\ \dfrac{\cdot}{\langle -1, \sigma_0\rangle \Downarrow \langle -1\rangle}}{\textsc{Assign}\ \dfrac{\langle x + (-1), \sigma_0\rangle \Downarrow \langle 1\rangle}{\langle S, \sigma_0\rangle \Downarrow \langle \sigma_1\rangle}}$$

Second iteration ($\sigma_1 \to \sigma_2$): the loop executes the second time, the value of x becomes 0. The condition of the loop:

$$\textsc{Less}\ \frac{\textsc{Const}\ \dfrac{\cdot}{\langle 0, \sigma_1\rangle \Downarrow \langle 0\rangle} \qquad \textsc{Lookup}\ \dfrac{\cdot}{\langle x, \sigma_1\rangle \Downarrow \langle 1\rangle}\ x \in \sigma_1}{\langle 0 < x, \sigma_1\rangle \Downarrow \langle \texttt{true}\rangle}$$

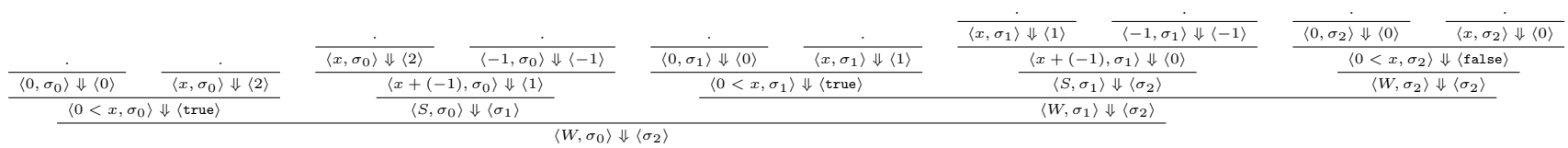
The first execution of the body of the loop:

$$\textsc{Add}\ \frac{\textsc{Lookup}\ \dfrac{\cdot}{\langle x, \sigma_1\rangle \Downarrow \langle 1\rangle}\ x \in \sigma_1 \qquad \textsc{Const}\ \dfrac{\cdot}{\langle -1, \sigma_1\rangle \Downarrow \langle -1\rangle}}{\textsc{Assign}\ \dfrac{\langle x + (-1), \sigma_1\rangle \Downarrow \langle 0\rangle}{\langle S, \sigma_1\rangle \Downarrow \langle \sigma_2\rangle}}$$

Loop termination (at σ_2): the condition in the loop is false in σ_2.

$$\textsc{While-False}\ \frac{\textsc{Less}\ \dfrac{\textsc{Const}\ \dfrac{\cdot}{\langle 0, \sigma_2\rangle \Downarrow \langle 0\rangle} \qquad \textsc{Lookup}\ \dfrac{\cdot}{\langle x, \sigma_2\rangle \Downarrow \langle 0\rangle}\ x \in \sigma_2}{\langle 0 < x, \sigma_2\rangle \Downarrow \langle \texttt{false}\rangle}}{\langle W, \sigma_2\rangle \Downarrow \langle \sigma_2\rangle}$$

A **complete derivation** is shown on the next page but, because it is quite big, the rule names are removed. However, it is intended to be viewed by zooming in rather than on paper, as the font size is quite small.


$\langle 0, \sigma_0 \rangle \Downarrow \langle 0 \rangle$
$\langle x, \sigma_0 \rangle \Downarrow \langle 2 \rangle$
$\langle 0 < x, \sigma_0 \rangle \Downarrow \langle \texttt{true} \rangle$
$\langle x, \sigma_0 \rangle \Downarrow \langle 2 \rangle$
$\langle -1, \sigma_0 \rangle \Downarrow \langle -1 \rangle$
$\langle x + (-1), \sigma_0 \rangle \Downarrow \langle 1 \rangle$
$\langle S, \sigma_0 \rangle \Downarrow \langle \sigma_1 \rangle$
$\langle 0, \sigma_1 \rangle \Downarrow \langle 0 \rangle$
$\langle x, \sigma_1 \rangle \Downarrow \langle 1 \rangle$
$\langle 0 < x, \sigma_1 \rangle \Downarrow \langle \texttt{true} \rangle$
$\langle x, \sigma_1 \rangle \Downarrow \langle 1 \rangle$
$\langle -1, \sigma_1 \rangle \Downarrow \langle -1 \rangle$
$\langle x + (-1), \sigma_1 \rangle \Downarrow \langle 0 \rangle$
$\langle S, \sigma_1 \rangle \Downarrow \langle \sigma_2 \rangle$
$\langle 0, \sigma_2 \rangle \Downarrow \langle 0 \rangle$
$\langle x, \sigma_2 \rangle \Downarrow \langle 0 \rangle$
$\langle 0 < x, \sigma_2 \rangle \Downarrow \langle \texttt{false} \rangle$
$\langle W, \sigma_2 \rangle \Downarrow \langle \sigma_2 \rangle$
$\langle W, \sigma_1 \rangle \Downarrow \langle \sigma_2 \rangle$
$\langle W, \sigma_0 \rangle \Downarrow \langle \sigma_2 \rangle$

2.3 Implementation Challenges

Implementing statement execution as a terminating function in Dafny is problematic due to potentially non-terminating while loops. The `decreases` clause cannot be provided for arbitrary loops.

```
function execStmt(stmt: Stmt, s: State): State
    decreases ? // <-- problem here
{
  match stmt {
    case Sgip => s
    case Assign(someVar, a) => s[someVar := evalAExp(a, s)]
    case Seq(s1, s2) => execStmt(s2, execStmt(s1, s))
    case If(b, s1, s2) => if evalBExp(b, s) then execStmt(s1, s) else
       execStmt(s2, s)
    case While(b, body) =>
      if evalBExp(b, s) then execStmt(While(b, body), execStmt(body, s))
         else s
  }
}
```

Solution: Relational Semantics with Gas. Instead of a function, we use a ghost predicate with a gas parameter to ensure termination:

```
ghost predicate evalStmt(stmt : Stmt, s : State, s1 : State, g:nat)
  decreases stmt, g
{
    match stmt
    case Skip => g == 1 && s == s1
    case Assign(myVar, ae) => g == 1 && s[myVar := evalAExp(ae, s)] == s1
    case Seq(stmt1, stmt2) =>
        exists g0 : nat, s_tmp : State ::
            0 < g0 < g &&
            evalStmt(stmt1, s, s_tmp, g0) &&
            evalStmt(stmt2, s_tmp, s1, g-g0)
    case If(b, stmt1, stmt2) =>
         if evalBExp(b,s)
         then evalStmt(stmt1, s, s1, g)
         else evalStmt(stmt2, s, s1, g)
    case While(b, body) =>
         if evalBExp(b, s)
         then exists s2: State , g0: nat ::
                0 < g0 < g &&
                evalStmt(body, s , s2 , g0) &&
                evalStmt(stmt, s2 , s1 , g-g0)
         else g == 1 && s1 == s
}
```

The gas parameter g decreases with each recursive call, ensuring termination of the predicate definition.

Verification Example. Using this relational semantics, we can prove that specific programs terminate with expected results. Below is a proof which corresponds to the derivation for the

program `while` $(0 < x)$ `do` $x := x + (-1)$ where $\sigma_0 = \{x \mapsto 2\}$.

```
lemma loop_to_zero ()
    ensures evalStmt ( While ( Less(Num(0) , Var(x)) ,
                                 Assign (x, Plus(Var(x) , Num( -1)) )
                             ) ,
                        map[x := 2] ,
                        map[x := 0] ,
                    3)
{
    var while_stmt :=  While ( Less(Num(0) , Var(x)) ,
                                 Assign (x, Plus(Var(x) , Num( -1)) )
                         );
    var cond := Less(Num(0) , Var(x));
    var body := Assign (x, Plus(Var(x) , Num( -1)) ) ;
    var aexp := Plus(Var(x) , Num( -1));
    var sigma := map[x := 2];
    var sigma1 := sigma[x := 1];
    var sigma2 := sigma1[x := 0];

    // calculational proof
    calc <== {
      evalStmt (while_stmt, sigma, sigma2, 3);

      evalBExp(cond, sigma) &&
          evalStmt(body, sigma , sigma1 , 1) &&
          evalStmt(while_stmt, sigma1 , sigma2 , 2);
          { assert evalAExp(aexp, sigma) == 1; }

      evalAExp(aexp, sigma) == 1 && evalStmt(while_stmt,sigma1,sigma2,2);

      evalBExp(cond, sigma1) &&
        evalStmt(body, sigma1, sigma2 , 1) &&
        evalStmt(while_stmt, sigma2, sigma2, 1);

      evalAExp(aexp, sigma1) == 0;

      true;
  }

  assert evalAExp(aexp, sigma1) == 0;
}
```

The lemma above proves that the while loop terminates in three steps, transforming the state from $\{x \mapsto 2\}$ to $\{x \mapsto 0\}$. In Dafny, a calculational proof (`calc <==  ...  `) shows a logical chain of reasoning where each step follows from the previous by inference rules or equalities. Here, it mirrors the operational semantics derivation tree of the while loop. The `evalStmt (while_stmt, sigma, sigma2, 3);` is the root of the derivation tree. The next row:

```
      evalBExp(cond, sigma) &&
          evalStmt(body, sigma , sigma1 , 1) &&
          evalStmt(while_stmt, sigma1 , sigma2 , 2);
          { assert evalAExp(aexp, sigma) == 1; }
```

captures the premisses for While-True.

The intermediary step below:

```
115       evalAExp(aexp, sigma) == 1 && evalStmt(while_stmt,sigma1,sigma2,2);
```

corresponds to the derivation of the arithmetic expression inside the assignment and the second evaluation of the loop body, which again, is justified by:

```
116       evalBExp(cond, sigma1) &&
117         evalStmt(body, sigma1, sigma2 , 1) &&
118         evalStmt(while_stmt, sigma2, sigma2, 1);
```

Finally, the arithmetic expression is evaluated to zero. The `assert` statements are necessary because they help guide the SMT solver that Dafny uses for verification by making intermediate facts explicit.