Operational Semantics

IMP Language and Big-Step Semantics

October 16, 2025

Overview

Formal Methods and Semantics

Formal methods provide instruments for defining and reasoning about programming languages.

The **semantics** of a language — the precise meaning of its constructs.

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 captured through logical assertions and proof rules

Focus of this presentation: Operational semantics (big-step)

Introducing IMP

IMP: A minimal imperative programming language

Why IMP?

- Includes fundamental control flow constructs (sequencing, conditionals, loops)
- Simple enough for clear, manageable formal definitions and proofs
- Expressive enough to write non-trivial programs
- Techniques extend naturally to more complex languages

Despite its simplicity, IMP captures the essential features of imperative programming.

IMP Syntax: BNF Grammar

$$n \in \mathbb{Z}$$

$$x \in \mathsf{Id}$$

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 \times a_2$$

$$b ::= \mathtt{true} \mid \mathtt{false} \mid a_1 < a_2 \mid \neg b \mid b_1 \wedge b_2$$

$$S ::= \mathtt{skip} \mid x := a \mid S_1; S_2 \mid \mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2 \mid \mathtt{while}\ b\ \mathtt{do}\ S$$

where:

- a ranges over **arithmetic expressions**
- b ranges over **boolean expressions**
- S ranges over **statements**

Identifiers in Dafny

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

We represent identifiers as an enumerated datatype for simplicity.

Alternatively, we could use strings.

Arithmetic Expressions in Dafny

BNF:

$$a ::= n \mid x \mid a_1 + a_2 \mid a_1 \times a_2$$

Dafny:

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

Examples:

- `Num(5)` represents the constant 5
- `Var(x)` represents the variable x
- `Plus(Var(x), Num(2))` represents $x + 2$

Boolean Expressions in Dafny

BNF:

$$b ::= \texttt{true} \mid \texttt{false} \mid a_1 < a_2 \mid \neg b \mid b_1 \wedge b_2$$

Dafny:

```
datatype BExp =
  | B(bool) // true or false, but this is easier
  | Less(AExp, AExp)
  | Not(BExp)
  | And(BExp, BExp)
```

Examples:

- `B(true)` represents the constant `true`
- `Less(Num(0), Var(x))` represents $0 < x$
- `Not(B(false))` represents $\neg$`false`

Statements in Dafny

BNF:

$$S ::= \texttt{skip} \mid x := a \mid S_1; S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2$$
$$\mid \texttt{while } b \texttt{ do } S$$

Dafny:

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

Examples:

- `Skip` does nothing
- `Assign(x, Num(5))` represents $x := 5$
- `Seq(s1, s2)` represents sequential composition $s_1; s_2$

States and Configurations

State: A partial function from variables to values

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

Configuration: All information needed to describe what a program is doing at a particular point in time.

For IMP, a configuration is a pair:

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

where σ is the program state and $\star \in \{\mathit{Stmt}, \mathit{AExp}, \mathit{BExp}, \mathit{Id}, \mathbb{Z}\}$.

Or simply $\langle * \rangle$ where $* \in \mathbb{Z}$ when state is irrelevant.

Configuration Examples

- An arithmetic expression configuration:

$$\langle x + 2, \{x \mapsto 5\}\rangle$$

- A boolean expression configuration:

$$\langle 0 < x, \{x \mapsto 3, y \mapsto 7\}\rangle$$

- A statement configuration (assignment):

$$\langle x := y + 1, \{y \mapsto 10\}\rangle$$

- A statement configuration (loop) :

$$\langle \texttt{while}\ (0 < x)\ \texttt{do}\ x := x + (-1), \{x \mapsto 2\}\rangle$$

- A final value configuration:

$$\langle 7 \rangle$$

States and Configurations in Dafny

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

- A `State` is a map from identifiers to integers (corresponds to partial functions $\sigma : \texttt{Id} \rightharpoonup \mathbb{Z}$)
- A `Configuration` is a pair of a statement and a state

Big-Step Semantics: Arithmetic Expressions

The notation

$$\langle a, \sigma \rangle \Downarrow \langle n \rangle$$

means that *"expression a evaluates to value n in state σ"*

Inference Rules:

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

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

Arithmetic Operations

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

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

Note: the + symbol is part of the syntax of IMP, but its semantics is given using $+_{\mathbb{Z}}$.

Evaluating Arithmetic Expressions 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)
  }
}
```

This function directly implements the inference rules.

Example: Arithmetic Derivation

Goal: Show that $\langle x + 2, \sigma \rangle \Downarrow \langle 4 \rangle$ where $\sigma(x) = 2$

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

This derivation tree use the inference rules for arithmetic expressions.

Big-Step Semantics: Boolean Expressions

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

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

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

Boolean Operations

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

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

Evaluating Boolean Expressions in Dafny

```
function evalBExp(b: BExp, s: State): bool {
  match b {
    case B(bval) => bval
    case Less(a1, a2) =>
      evalAExp(a1, s) < evalAExp(a2, s)
    case Not(b1) => !evalBExp(b1, s)
    case And(b1, b2) =>
      evalBExp(b1, s) && evalBExp(b2, s)
  }
}
```

Example: Boolean Derivation

Goal: Show that $\langle 0 < x, \sigma \rangle \Downarrow \langle \texttt{true} \rangle$ where $\sigma(x) = 2$

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

Big-Step Sem. Statements (1): skip, assignment, sequence

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

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

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

Big-Step Sem. Statements (2): decisional statements

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

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

Big-Step Sem. Statements (3): while loop

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

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

Note: The while rule is recursive; it allows non-terminating loops

Example Program

Consider the program:

$$\texttt{while } (0 < x) \texttt{ do } x := x + (-1)$$

Initial state: $\sigma_0 = \{x \mapsto 2\}$

Goal: Show that execution terminates in state $\sigma_2 = \{x \mapsto 0\}$

Notation:

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

First Iteration: Condition

First iteration $(\sigma_0 \to \sigma_1)$: Loop executes for the first time, x decreases by 1.

The condition of the loop:

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

First Iteration: Body

The first execution of the body: $\sigma_1 = \sigma_0[x \mapsto 1]$.

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

Second Iteration: Condition

Second iteration $(\sigma_1 \to \sigma_2)$: Loop executes again, x becomes 0.

The condition of the loop:

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

Second Iteration: Body

The second execution of the body: $\sigma_2 = \sigma_1[x \mapsto 0]$

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

Loop Termination

Loop termination (at σ_2): The condition is now false.

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

Implementation Challenges

Problem: Implementing statement evaluation as a (terminating) function in Dafny is problematic due to potentially non-terminating while loops.

The `decreases` clause cannot be provided for arbitrary loops.

We need a different approach!

Naive Implementation: non-termination issue

```
function execStmt(stmt: Stmt, s: State): State
    decreases ? // <-- cannot provide a decreases clause
{
  match stmt {
    case Skip => 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
  }
}
```

Issue: Cannot prove termination for `While` case!

Solution: define a relation instead of a function

Instead of a function, we use a ghost predicate with a gas parameter:

- The **gas parameter** $g : \mathbb{N}$ decreases with each recursive call
- Ensures termination of the predicate definition
- Represents a bound on computation steps when the statement under evaluation does not structurally decrease
- Predicate: $\texttt{evalStmt}(stmt, s, s', g)$ – "*stmt* evolves from state s to s' in at most g steps"

This approach separates:

- Semantics (what the program means)
- Termination (whether computation halts)

Relational Semantics: Skip and Assign

```
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
...
```

- `Skip`: Requires exactly 1 unit of gas, state unchanged
- `Assign`: Requires exactly 1 unit of gas, updates state

Relational Semantics: Seq and If

```
...
    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)
...
```

- Seq: Splits gas between two statements via intermediate state
- `If`: Uses full gas for chosen branch based on condition

Relational Semantics: While

```
...
    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
}
```

- If condition is `true`: execute body, then recursively evaluate loop with remaining gas
- If condition is `false`: terminate immediately (1 gas unit)
- Gas decreases, ensuring termination of the predicate

Verifying the While Loop

We can now prove that specific programs terminate with expected results!

The program gets executed directly using the formal semantics.

Goal:
Prove that $\texttt{while}\ (0 < x)\ \texttt{do}\ x := x + (-1)$ starting from $\sigma_0 = \{x \mapsto 2\}$ terminates in $\sigma_2 = \{x \mapsto 0\}$ using 3 gas units.

The proof in the next slides corresponds to the derivation we constructed earlier.

Verification Lemma (1/3)

```
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];
...
```

Verification Lemma (2/3)

```
...
    // 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);
...
```

The `calc <==` construct shows logical chain of reasoning.
Each step follows from the previous by inference rules.

Verification Lemma (3/3)

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

      evalAExp(aexp, sigma1) == 0;

      true;
  }

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

Key insight: The calculational proof mirrors the operational semantics derivation tree structure!

Understanding Calculational Proofs

What does `calc <== { ... }` do?

A calculational proof shows a logical chain where each step follows from the previous.

- `evalStmt (while_stmt, sigma, sigma2, 3);`
  Root of derivation tree
- `evalBExp(cond, sigma) && evalStmt(body, ...) && evalStmt(while_stmt, ...)`
  Captures premises for While-True rule
- Intermediate assertions guide the SMT solver
- Each line represents applying an inference rule

Takeaways

Operational Semantics:

- Defines meaning through execution steps
- Uses inference rules to specify evaluation
- Provides a foundation for reasoning about programs

Big-Step Semantics:

- Relates initial and final configurations directly
- Compositional: meaning of compound constructs from parts
- Can be formalized and verified in Dafny
- Programs can be **executed** using formal operational semantics
- Sometimes, the interpreter generated from an operational semantics is not fast enough, but it can serve as a *reference implementation* for compilers or interpreters

Implementation Strategy

Challenges:

- While loops may not terminate
- Cannot write direct recursive functions
- Need to ensure termination of definitions

Solutions:

- Use relational semantics (predicates, not functions)
- Add gas parameter to bound recursion
- Separate semantics from termination concerns
- Verify specific programs with explicit gas amounts

This approach extends to more complex languages and properties!

From Theory to Practice

What we've achieved:

1. Formal specification of IMP syntax
2. Big-step operational semantics via inference rules
3. Implementation in Dafny with verification support
4. Concrete example: verified while loop execution

Next steps:

- Axiomatic semantics (Hoare Logic)
- Program verification techniques
- Proving program properties (correctness, termination)

Connections to Broader Topics

- **Program verification:** Proving programs satisfy specifications
- **Compiler correctness:** Showing optimizations preserve meaning
- **Language design:** Precise understanding of new features
- **Static analysis:** Sound approximations of program behavior
- Better **software engineering:** Rigorous engineering practices

Operational semantics provides the mathematical foundation for reasoning about what programs *actually do*.

Questions?

Thank you for your attention!