Programming Language Semantics

An overview of operational, denotational and axiomatic styles of semantics

Dhruv Makwana

28th January, 2020

These notes aim to provide a very brief overview of three styles of formal semantics used to describe and understand programming languages, by defining the syntax and semantics of a small, imperative language called 'IMP'. They borrow heavily from Winskel [1993] for the structure and examples.

1.1 Syntax of IMP

$$\begin{array}{lll} a & ::= & n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1 \\ \\ b & ::= & \textbf{true} \mid \textbf{false} \mid a_0 == a_1 \mid a_0 \leq a_1 \mid \,!\, b \mid b_0 \,\&\&\, b_1 \mid b_0 \,||\, b_1 \\ \\ c & ::= & \textbf{skip} \mid X = a \mid c_0; c_1 \mid \textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1 \\ & & \textbf{while}\ b\ \textbf{do}\ c \end{array}$$

1. No functions (for simplicity's sake).
2. All terms are "well-typed" by definition.
3. In code: either tagged-unions or inheritance.

What is the point of defining our own, very simple language instead of using real-world languages familiar to us regular programmers?

First, it allows us to isolate the precise features we wish to study under the sterile environment of mathematics (one may draw an analogy with laboratory experiments done in the physical sciences); as we understand more, our languages of study can become more complex and realistic.

Secondly, even this simple language is Turing complete, so we don't lose any theoretical expressivity.

1.2 Operational Semantics of IMP

Operational semantics are an abstract, mathematical specification of an interpreter.[1]

[1] The style presented here is big-step semantics.

Because of their abstract nature, these specifications can be a bit terse. However, they are vastly preferred (by researchers and language-implementers) to natural-language semantics because there is no room for ambiguity.

A key skill as a working semanticist is to be able to take descriptions like the ones below and translate them into a working (ideally correct, and possibly even efficient) implementations. Here, we use Java as one example language; however typed-functional languages are ideal for implementing such rules.

Evaluation of Arithmetic Expressions

$\langle a, \sigma \rangle \to n$ specifies an *evaluation function*, from a pair of an arithmetic expression a and a state σ (finite map from variables to integers) to an integer n.

$$\frac{}{\langle n, \sigma \rangle \to n}$$

$$\frac{}{\langle X, \sigma \rangle \to \sigma(X)} \text{ if } X \in \mathrm{dom}(\sigma)$$

$$\frac{\langle a_0, \sigma \rangle \to n_0 \qquad \langle a_1, \sigma \rangle \to n_1}{\langle a_0 + a_1, \sigma \rangle \to n_{\mathrm{sum}}} \; n_{\mathrm{sum}} = n_0 + n_1$$

- *This does not specify an evaluation order.*
- *Behaviour is undefined if variable is not in state.*

```
class AddExpr extends ArithExpr {
  ArithExpr left, right;
  @Override
  public int eval(Map<Var,Int> state) {
    return right.eval(state) + left.eval(state);
  }
}
```

Specification also allows left-first or parallel evaluation.

Under-specified and undefined behaviour are perfectly acceptable. In fact, both are crucial for optimising compilers. An optimisation is an analysis and transformation of code. Under-specified or undefined behaviour allow the complier to make more assumptions to enable more valid transformations.

In the above example, we could even evaluate the left and right sub-expressions in parallel. Or, when implementing variable-lookup, the implementation could skip null-checks, thus transferring the responsibility of assign-before-use onto the programmer.

Evaluation of Boolean Expressions

$\langle b, \sigma \rangle \rightarrow v$ specifies an *evaluation function*, from a pair of a boolean expression b and a state σ to a boolean v.

$$\frac{\langle b_0, \sigma \rangle \rightarrow \textbf{false}}{\langle b_0 \;\&\&\; b_1, \sigma \rangle \rightarrow \textbf{false}}$$

$$\frac{\langle b_0, \sigma \rangle \rightarrow \textbf{true} \qquad \langle b_1, \sigma \rangle \rightarrow v}{\langle b_0 \;\&\&\; b_1, \sigma \rangle \rightarrow v}$$

THIS *forces* A LEFT-TO-RIGHT EVALUATION ORDER.

```
class AndExpr extends BoolExpr {
  BoolExpr left, right;
  @Override
  public boolean eval(Map<Var,Int> state) {
    return left.eval(state) && right.eval(state);
  }
}
```

Evaluation of Commands

$\langle c, \sigma \rangle \rightarrow \sigma'$ specifies an *evaluation function*, from a pair of a command c and a state σ to a state σ'. Read $\sigma + \{X \mapsto n\}$ as "update key X in map σ with value n".

$$\frac{}{\langle \textbf{skip}, \sigma \rangle \rightarrow \sigma}$$

$$\frac{\langle a, \sigma \rangle \rightarrow n}{\langle X = a, \sigma \rangle \rightarrow \sigma + \{X \mapsto n\}}$$

$$\frac{\langle c_0, \sigma \rangle \rightarrow \sigma' \qquad \langle c_1, \sigma' \rangle \rightarrow \sigma''}{\langle c_0; c_1, \sigma \rangle \rightarrow \sigma''}$$

```
class AssignCmd extends Command {
  Var var;
  ArithExpr arith;
  @Override
  public Map<Var,Int> eval(Map<Var,Int> state) {
    return state.put(var, arith.eval(state));
  }
}
```

$$\frac{\langle b, \sigma\rangle \rightarrow \textbf{true} \qquad \langle c_0, \sigma\rangle \rightarrow \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \rightarrow \sigma'}$$

$$\frac{\langle b, \sigma\rangle \rightarrow \textbf{false} \qquad \langle c_1, \sigma\rangle \rightarrow \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1, \sigma\rangle \rightarrow \sigma'}$$

$$\frac{\langle b, \sigma\rangle \rightarrow \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \rightarrow \sigma}$$

$$\frac{\langle b, \sigma\rangle \rightarrow \textbf{true} \quad \langle c, \sigma\rangle \rightarrow \sigma' \quad \langle \textbf{while } b \textbf{ do } c, \sigma'\rangle \rightarrow \sigma''}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \rightarrow \sigma''}$$

Unlike expressions, commands do not evaluate to a value, modifying state (the store) instead. Commands are typically used for side-effects, such as memory modification, IO and network access.

Let $w \equiv \textbf{while } b \textbf{ do } c$. Define $c_0 \sim c_1$ as $\forall \sigma, \sigma'.\ \langle c_0, \sigma\rangle \rightarrow \sigma' \Leftrightarrow \langle c_1, \sigma\rangle \rightarrow \sigma'$.

Theorem 1.1. $w \sim \textbf{if } b \textbf{ then } \{c; w\} \textbf{ else skip}$.

What do we gain by defining and proving such properties? We gain the ability to precisely state and verify intuitive assumptions about a language and its evaluation strategy.

We can take for-loops as an example: imagine we want to add for-loops to IMP. We could extend the existing interpreter to support the new construct and its evaluation rules. Or, we could (a) define the operational semantics of a for-loop (b) define its translation into other, pre-existing constructs (if-statements and while-loops) and (c) check that the evaluation of the for-loop matches the evaluation of the translated one. Then, all we would need to do is add a simple syntactic transformation after parsing: we would have no need to touch the interpreter.

1.3 Denotational Semantics

Denotational semantics defines the *meaning* of programs in a *syntax-independent* way, in terms of a well-understood area of mathematics (domain theory or category theory). This allows us to reason about *program equivalence* more generically, potentially across *different programming languages.*

Denotations of Arithmetic & Boolean Expressions

$\mathcal{A}[\![a]\!]$ and $\mathcal{B}[\![b]\!]$ are *functions* from a state, to a number and boolean respectively.

$$\begin{aligned}
\mathcal{A}[\![n]\!] &= \lambda\sigma.\ n \\
\mathcal{A}[\![X]\!] &= \lambda\sigma.\ \sigma(X) \\
\mathcal{A}[\![a_0 + a_1]\!] &= \lambda\sigma.\ \mathcal{A}[\![a_0]\!]\sigma + \mathcal{A}[\![a_1]\!]\sigma \\
\mathcal{B}[\![b_0\ \&\&\ b_1]\!] &= \lambda\sigma.\ \begin{array}{ll} \text{true} & \mathcal{B}[\![b_0]\!]\sigma = \text{true and } \mathcal{B}[\![b_1]\!]\sigma = \text{true} \\ \text{false} & \text{otherwise} \end{array}
\end{aligned}$$

In the absence of side-effects in expressions, order of evaluation specified *operationally* is irrelevant.

If you have a mathematical background, this presentation is likely to be more pleasing.

Programs are said to denote *functions* from state to a value. This means we can ignore (some) details of *how* a program is executed and focus more on *what* it is computing.

This also allows us to use more sophisticated mathematics in manipulating the functions our programs represent.

Denotations of Simple Commands

For simple commands, the denotations are straightforward functions from state to state.

$$\begin{aligned}
\mathcal{C}[\![\textbf{skip}]\!] &= \lambda\sigma.\ \sigma \\
\mathcal{C}[\![X = a]\!] &= \lambda\sigma.\ \sigma + \{X \mapsto \mathcal{A}[\![a]\!]\sigma\} \\
\mathcal{C}[\![c_0; c_1]\!] &= \mathcal{C}[\![c_1]\!] \circ \mathcal{C}[\![c_0]\!] \\
\mathcal{C}[\![\textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1]\!] &= \lambda\sigma.\ \begin{array}{ll} \mathcal{C}[\![c_0]\!]\sigma & \text{if } \mathcal{B}[\![b]\!]\sigma = \text{true} \\ \mathcal{C}[\![c_1]\!]\sigma & \text{otherwise} \end{array}
\end{aligned}$$

Subjectively, I think there is something rather elegant and beautiful in denoting the sequencing-semicolon as function composition.

Problems with Denotation of While-loop

There are constraints that the definition of denotations must follow, which make it problematic to handle while-loops:

- Must be *compositional*: only the meaning of the parts determines the meaning of the whole.
- Cannot be *arbitrarily* self-referential for the sake of mathematical consistency (viz. ZFC & Russel's paradox).

- Must be able to represent and propagate (non-)termination (viz. halting problem).

Why must denotations be compositional? It greatly simplifies proofs if they are – we don't need to consider special cases of particular combinations of syntax (think of puzzlers in your favourite language). A simple example from Slang illustrates the importance of this: $[\![A += B]\!] \neq [\![A = A + B]\!]$.[2]

[2] Example in `Test: Slang Gotchas`

Domain theory[3] provides both

[3] Modern semanticists rely heavily on category theory.

- Least-upper-bounds (suprema) for constructing *fixed-points* to represent recursion *soundly.*
- *Continuous functions* for preserving and propagating termination.

Continuity in domain theory is a generalisation of what you may have seen in an introduction to analysis class. It includes montonocity and the preservation of least-upper-bounds.

In this context, monotonicity ensures we can't construct functions of the form "if *this* doesn't terminate then do *this other thing* otherwise do *that*" and preservation of least-upper-bounds ensures we can only construct functions that preserve termination in a sensible manner.

Denotation of While-loop

$$\mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] = \text{fix}(\Gamma)$$

where

$$\Gamma(\phi) = \lambda\sigma.\ \begin{array}{ll} \sigma & \text{if } \mathcal{B}[\![b]\!]\sigma = \text{false} \\ \phi(\mathcal{C}[\![c]\!]\sigma) & \text{otherwise} \end{array}$$

and

$$\text{fix}(f) = \bigsqcup_{n \in \mathbb{N}} f^n(\bot)$$

You can think of Γ's argument ϕ as factoring out the recursion; it's the mathematical equivalent of passing in a callback.

Explaining 'fix'

$$\Gamma^0(\bot) = \lambda\sigma.\ \bot$$

$$\Gamma^1(\bot) = \lambda\sigma.\ \begin{cases} \sigma & \text{if } \mathcal{B}[\![b]\!]\sigma = \text{false} \\ \bot & \text{otherwise} \end{cases}$$

$$\Gamma^2(\bot) = \lambda\sigma.\ \begin{cases} \sigma & \text{if } \mathcal{B}[\![b]\!]\sigma = \text{false} \\ \mathcal{C}[\![c]\!]\sigma & \text{if } \mathcal{B}[\![b]\!]\sigma = \text{true and} \\ & \mathcal{B}[\![b]\!](\mathcal{C}[\![c]\!]\sigma) = \text{false} \\ \bot & \text{otherwise} \end{cases}$$

$$\vdots$$

The intuition for 'fix' is a little more subtle: $\bot$ represents the callback which never terminates. Every time we iterate, we effectively add an extra branch to this piecewise function. By taking the big-union over all of the functions, we allow the possibility of the loop executing any number of times.

Equivalence of Operational and Denotational Semantics

Because we invented our operational rules 'out of thin air', as manipulation of pure syntax, it's important to justify those rules using well-known maths (especially the self-referential semantics of a while-loop).

Theorem 1.2. $\forall \sigma, \sigma'.\ \langle c, \sigma\rangle \to \sigma' \Leftrightarrow \mathcal{C}[\![c]\!]\sigma = \sigma'$

1.4 Axiomatic Semantics

So far, we've been looking at proving properties about the execution and meanings of *all* programs. But what if we want to understand a *particular* program?

$$S = 0; N = 0;$$

$$\textbf{while } !(N == 101) \textbf{ do}$$

$$\{S = S + N; N = N + 1\}$$

How would we prove that this program, if it terminates, ends with the value of S as 5050 ($\sum_{m=1}^{100} m$)?

Syntax of IMP Assertions

We need to define an *assertion-language* (with its own syntax and semantics!) so that we can precisely state properties we want to prove.

$$\begin{array}{lcl} a & ::= & n \mid X \mid i \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \\ \\ A & ::= & \mathbf{true} \mid \mathbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg A \mid A_0 \wedge A_1 \\ & & A_0 \vee A_1 \mid A_0 \Rightarrow A_1 \mid \forall i.\ A \mid \exists i.\ A \end{array}$$

Though the notion of assertion-langauges might sound strange, we can map it back to what we've understood here. A proof would be a valid program in the assertion-language; an operational semantics corresponds to proof simplification; a denotational semantics would justify the consistency of our assertion-langauge. However, all of this is *far* beyond the scope of these notes.

Generating Assertions (1)

$$\frac{}{\{A\}\mathbf{skip}\{A\}}$$

$$\frac{\{A\}c_0\{B\} \quad \{B\}c_1\{C\}}{\{A\}c_0; c_1\{C\}}$$

$$\frac{\{A \wedge b\}c_0\{B\} \quad \{A \wedge \neg b\}c_1\{B\}}{\{A\}\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1\{B\}}$$

Why do we want to *generate* assertions? Automatic code analysis tools rely on this to figure out which pre-conditions might not be met by a particular piece of a program. For example, a method-call on an object in Java has the pre-condition that the object is not null.

Generating Assertions (2)

Read $B[a/X]$ as "substitute a for X in B".

$$\frac{}{\{B[a/X]\}X = a\{B\}}$$

$$\frac{\{A \wedge b\}c\{A\}}{\{A\}\mathbf{while}\ b\ \mathbf{do}\ c\{A \wedge \neg b\}}$$

$$\frac{A \Rightarrow A' \quad \{A'\}c_0\{B'\} \quad B' \Rightarrow B}{\{A\}c\{B\}}$$

The assignment rule might look backwards; it isn't. To convince yourself of this, consider $\{X + 1 = 2\} X = X + 1 \{X = 2\}$. With the substitution the other way around, your logic would be inconsistent; you could prove $\{X = 0\} X = 1 \{1 = 0\}$.

Proving a program correct

Theorem 1.3. *The sum program is correct w.r.t its specification: key step is loop invariant $S = \sum_{m=1}^{N-1} m$.*

1.5 Conclusion

- Different ways of understanding languages and programs.
- Material presented here is circa 1970's research, well-established by Winskel [1993] – can handle realistic languages and proofs now.
- All programmers already have an intuitive understanding of these things – but this gives more precision to our thoughts.

References

Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT press, 1993.