Programming Language Semantics

An overview of operational, denotational and axiomatic styles of semantics

Dhruv Makwana

28th January, 2020

TenMinute Tech-Net
Engineering, Goldman Sachs

Syntax of IMP

Syntax of IMP

$$
\begin{array}{rcl}
a & ::= & n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1 \\[2ex]
b & ::= & \mathbf{true} \mid \mathbf{false} \mid a_0 == a_1 \mid a_0 \leq a_1 \mid !\, b \mid b_0 \,\&\&\, b_1 \mid b_0 \mid\mid b_1 \\[2ex]
c & ::= & \mathbf{skip} \mid X = a \mid c_0 ; c_1 \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \\
 & & \mathbf{while}\ b\ \mathbf{do}\ c
\end{array}
$$

Syntax of IMP

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

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

Operational Semantics of IMP

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

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

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.

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

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

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

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.

Evaluation of Arithmetic Expressions

$\langle a, \sigma \rangle \rightarrow n$ 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.

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

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

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

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

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.

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

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.

```
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.

Evaluation of Boolean Expressions

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

$$\frac{\langle b_0, \sigma \rangle \to \mathbf{false}}{\langle b_0 \mathbin{\&\&} b_1, \sigma \rangle \to \mathbf{false}}$$

$$\frac{\langle b_0, \sigma \rangle \to \mathbf{true} \qquad \langle b_1, \sigma \rangle \to v}{\langle b_0 \mathbin{\&\&} b_1, \sigma \rangle \to v}$$

Evaluation of Boolean Expressions

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

$$\frac{\langle b_0, \sigma \rangle \to \mathbf{false}}{\langle b_0 \mathbin{\&\&} b_1, \sigma \rangle \to \mathbf{false}}$$

$$\frac{\langle b_0, \sigma \rangle \to \mathbf{true} \qquad \langle b_1, \sigma \rangle \to v}{\langle b_0 \mathbin{\&\&} b_1, \sigma \rangle \to v}$$

This *forces* a left-to-right evaluation order.

Evaluation of Boolean Expressions

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

```
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

$\langle c, \sigma \rangle \to \sigma'$ 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".

$$\frac{}{\langle \mathbf{skip}, \sigma \rangle \to \sigma}$$

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

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

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'.

```
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));
  }
}
```

Evaluation of Commands

$\langle c, \sigma \rangle \to \sigma'$ specifies an *evaluation function*, from a pair of a command c and a state σ to a state σ'.

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

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

$$\frac{\langle b, \sigma \rangle \to \mathbf{false}}{\langle \mathbf{while}\ b\ \mathbf{do}\ c, \sigma \rangle \to \sigma}$$

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

Let $w \equiv \mathbf{while}\ b\ \mathbf{do}\ c$.

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

$w \sim \mathbf{if}\ b\ \mathbf{then}\ \{c; w\}\ \mathbf{else\ skip.}$

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.

Denotations of Arithmetic & Boolean Expressions

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

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

Denotations of Arithmetic & Boolean Expressions

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

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

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

Denotations of Simple Commands

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

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

Problems with Denotation of While-loop

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

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.

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).

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).

Domain Theory

Domain theory[2] provides both

[2]Modern semanticists rely heavily on category theory.

Domain Theory

Domain theory[2] provides both

- Least-upper-bounds (suprema) for constructing *fixed-points* to represent recursion *soundly*.

[2] Modern semanticists rely heavily on category theory.

Domain Theory

Domain theory[2] provides both

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

[2] Modern semanticists rely heavily on category theory.

Denotation of While-loop

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

where

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

and

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

Explaining ‘fix’

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

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

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

$$\vdots$$

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

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

Axiomatic Semantics

So far, we’ve been looking at proving properties about the execution and meanings of *all* programs.

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?

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.

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.

$$
\begin{aligned}
a \quad &::= \quad n \mid X \mid i \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 \times a_1 \\[1em]
A \quad &::= \quad \mathbf{true} \mid \mathbf{false} \mid a_0 = a_1 \mid a_0 \leq a_1 \mid \neg A \mid A_0 \wedge A_1 \\
&\qquad\; A_0 \vee A_1 \mid A_0 \Rightarrow A_1 \mid \forall i.\ A \mid \exists i.\ A
\end{aligned}
$$

Generating Assertions (1)

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

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

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

Generating Assertions (2)

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

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

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

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

Proving a program correct

Theorem

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

Conclusion

- Different ways of understanding languages and programs.

- 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.

- 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.

Thank you