Introduction to Theoretical Computer Science

Lecture 18: Denotational Semantics

Dr. Liam O'Connor

LFCS, University of Edinburgh
CECS, Australian National University
Semester 1, 2023/2024

Semantics

This lecture concerns the topic of *semantics*, which is a mathematical description of the meaning of programs.

Why learn this?

We can't prove anything about a computer program without first giving it a semantics.

Semantics

Semantics can be specified in many ways:

1. *Denotational Semantics* is the *compositional* construction of a *mathematical object* for each form of *syntax*. MCS
2. *Axiomatic Semantics* is the construction of a *proof calculus* to allow correctness of a program to be verified. AR, FV
3. *Operational Semantics* is the construction of a program-evaluating *state machine* or *transition system*. TSPL, EPL, MCS

In this lecture

We focus mostly on denotational semantics as MCS's treatment is very informal and no other course touches it.

Denotational Semantics

At its heart, it's quite simple:

$$[\![\cdot]\!] : \text{Program} \to \text{Semantics}$$

More specifically, we define a function $[\![\cdot]\!]$ which maps *syntax* into (mathematical) *models*.

Desideratum

We want this semantic function to be *compositional*: The semantics of a compound expression should be made from the semantics of its components.

Robot Example

Example (A Toy Language)

A robot moves along a grid according to a sequence of commands `move` (forward 1 unit) and `turn` (90 degrees counter-clockwise), separated by semicolons, with the command sequence terminated by the keyword `stop`:

$$\mathcal{R} ::= \texttt{move};\ \mathcal{R} \mid \texttt{turn};\ \mathcal{R} \mid \texttt{stop}$$

$$\begin{aligned}
[\![\cdot]\!]^{\mathcal{R}} &: \mathcal{R} \to \mathbb{Z}^2 \\
[\![\texttt{turn};\ r]\!]^{\mathcal{R}} &= \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} [\![r]\!]^{\mathcal{R}} \\
[\![\texttt{move};\ r]\!]^{\mathcal{R}} &= [\![r]\!]^{\mathcal{R}} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
[\![\texttt{stop}]\!]^{\mathcal{R}} &= \begin{pmatrix} 0 \\ 0 \end{pmatrix}
\end{aligned}$$

Arithmetic Expressions

$$[\![\cdot]\!]^{\mathcal{E}} : \mathcal{E} \to \Sigma \to \mathbb{Z}$$

Our denotation for arithmetic expressions is functions from *states* (mapping from variables to their values) to values.

$$\begin{aligned}
[\![n]\!]^{\mathcal{E}} &= \lambda\sigma.\ n \\
[\![x]\!]^{\mathcal{E}} &= \lambda\sigma.\ \sigma(x) \\
[\![e_1 \texttt{ + } e_2]\!]^{\mathcal{E}} &= \lambda\sigma.\ [\![e_1]\!]^{\mathcal{E}}\sigma + [\![e_2]\!]^{\mathcal{E}}\sigma \\
[\![e_1 \texttt{ * } e_2]\!]^{\mathcal{E}} &= \lambda\sigma.\ [\![e_1]\!]^{\mathcal{E}}\sigma \times [\![e_2]\!]^{\mathcal{E}}\sigma \\
[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]^{\mathcal{E}} &= \lambda\sigma.\ [\![e_2]\!]^{\mathcal{E}} \left(\sigma[x := [\![e_1]\!]^{\mathcal{E}}\sigma]\right)
\end{aligned}$$

Where $\sigma[x := n]$ is a new state just like σ except the variable x now maps to n.

Note: From this point onwards I'll assume all standard arithmetic expressions are in $\mathcal{E}$

Boolean Expressions

$$[\![\cdot]\!]^{\mathcal{B}} : \mathcal{B} \to \mathcal{P}(\Sigma)$$

Our denotation for a boolean expression is a set of *states* that satisfy the predicate represented by the expression.

$$\begin{array}{lcl}
[\![e_1 \texttt{ == } e_2]\!]^{\mathcal{B}} & = & \{\sigma \mid [\![e_1]\!]^{\mathcal{E}}\sigma = [\![e_2]\!]^{\mathcal{E}}\sigma\} \\
[\![e_1 \texttt{ <= } e_2]\!]^{\mathcal{B}} & = & \{\sigma \mid [\![e_1]\!]^{\mathcal{E}}\sigma \leq [\![e_2]\!]^{\mathcal{E}}\sigma\} \\
[\![e_1 \texttt{ \&\& } e_2]\!]^{\mathcal{B}} & = & [\![e_1]\!]^{\mathcal{B}} \cap [\![e_2]\!]^{\mathcal{B}} \\
[\![e_1 \texttt{ || } e_2]\!]^{\mathcal{B}} & = & [\![e_1]\!]^{\mathcal{B}} \cup [\![e_2]\!]^{\mathcal{B}} \\
[\![\texttt{! } e_1]\!]^{\mathcal{B}} & = & \Sigma \setminus [\![e_1]\!]^{\mathcal{B}}
\end{array}$$

Note: C notation is used here to distinguish syntax from semantics, but from this point onwards I'll assume all standard boolean expressions are in $\mathcal{B}$

Imperative Programs

We are going to give semantics to non-deterministic imperative programs. Because of non-determinism, our models are relations not functions:

$$[\![\cdot]\!] : \mathcal{I} \to \mathcal{P}(\Sigma \times \Sigma)$$

$(\sigma_1, \sigma_2) \in [\![P]\!]$ means that executing P on an initial state σ_1 may result in the final state σ_2.

Assignment statement

An *assignment* $x := e$ simply assigns the value of the expression e to the variable x:

$$[\![x := e]\!] = \left\{ (\sigma_i, \sigma_f) \mid \sigma_f = \sigma_i \left[x \mapsto [\![e]\!]^{\mathcal{E}}(\sigma_i) \right] \right\}$$

More Statements

Sequencing

The semicolon, or *sequential composition* operator, is the operator that lets us first run P, and then run Q.

$$[\![P; Q]\!] = [\![P]\!] \fatsemi [\![Q]\!]$$

where $\fatsemi$ is forward-composition of relations:

$$X \fatsemi Y = \{(\sigma_i, \sigma_f) \mid \exists \sigma_m. (\sigma_i, \sigma_m) \in X \land (\sigma_m, \sigma_f) \in Y\}$$

Example (Swap)

$$(\{a \mapsto 4, b \mapsto 8, \ldots\}, \{a \mapsto 8, b \mapsto 4, \ldots\}) \\ \in [\![x := a; a := b; b := x]\!]$$

More Statements

Choice and Guards

An *a nondeterministic choice* $P + Q$ means that all observations of P and all observations of Q are possible:

$$[\![P + Q]\!] = [\![P]\!] \cup [\![Q]\!]$$

A boolean expression *guard* φ (in $\mathcal{B}$) doesn't change the state, but only those observations that satisfy φ succeed:

$$[\![\varphi]\!] = \left\{(\sigma, \sigma) \mid \sigma \in [\![\varphi]\!]^{\mathcal{B}}\right\}$$

Using these ingredients, we can recover **if**-statements:

$$\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q \textbf{ fi} \simeq (\varphi; P) + (\neg\varphi; Q)$$

Loops

the **skip** statement does nothing: $[\![\textbf{skip}]\!] = I = \{(\sigma, \sigma) \mid \sigma \in \Sigma\}$

Star

The *Kleene star* $P^\star$ is the operator that runs loop body P for a nondeterministic amount of times. The semantics are the smallest solution to this recursive equation:

$$[\![P^\star]\!] = I \cup [\![P]\!] \mathbin{;} [\![P^\star]\!] \quad \text{(i.e.} \quad P^\star \simeq \textbf{skip} + (P; P^\star)\text{)}$$

We will show that this is the same as:

$$[\![P^\star]\!] = \bigcup_{i \in \mathbb{N}_0} [\![P]\!]^i$$

Where superscripting is self-composition: $\begin{aligned} R^0 &= I \\ R^{n+1} &= R \mathbin{;} R^n \end{aligned}$

We can recover **while** loops: $\textbf{while } g \textbf{ do } P \textbf{ od} \simeq (g; P)^\star; \neg g$

Great Scott!

Rewriting our equation slightly:

$$[\![P^\star]\!] = f([\![P^\star]\!]) \text{ where } f(X) = I \cup [\![P]\!] \mathbin{;} X$$

A solution to this equation is a *fixed point* of the function f, i.e., a value x such that $f(x) = x$

1. Why does this equation have a solution?
2. If it has more than one solution, which one do we pick?

ω-cpos

We'll put our models into a *partial order* $\sqsubseteq$, read "approximates", which is an *ω-complete partial order*:

1. *Pointed*: it has a least element $\bot$ which approximates everything.
2. *ω-chain-complete*: For every countable ascending sequence $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \ldots$ we have a least upper bound, written $\sup f$ or $\bigsqcup_{n \in \mathbb{N}} f_n$.

Examples of cpos

- $(\mathcal{P}(S), \subseteq)$is a cpo: the LUB of a chain is just the union of the chain.
- $(\mathbb{N}, \leq)$ is not a cpo: $1 \leq 2 \leq 3 \leq \ldots$ has no LUB.
- $(\mathbb{N} \cup \{\infty\}, \leq)$ is a cpo, as ∞ is the LUB of any non-repeating chain.
- $(S, =)$is a *discrete domain*, which is a cpo.
- $(S_\perp, \sqsubseteq)$, i.e., the set S extended with a single least element $\perp$is a *flat domain*, which is a cpo.

In our case

Our cpo is $(\mathcal{P}(\Sigma \times \Sigma), \subseteq)$.

- The least element $\perp = \emptyset$
- The least upper bound of a chain $f_0 \subseteq f_1 \subseteq f2 \ldots$ is just $\bigcup_{i \in \mathbb{N}} f_i$

Climbing Chains

Recalling our semantics for the star operator, we want to show that the least fixed point of a function f on our cpo is the least upper bound of the *ascending Kleene chain*:

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq f^3(\bot) \sqsubseteq f^4(\bot) \sqsubseteq \cdots$$

But!

This chain doesn't exist for some f! Consider this f on the flat domain $(\mathbb{N}_\bot, \sqsubseteq)$:

$$f(x) = \begin{cases} 1 & \text{if } x = \bot \\ \bot & \text{if } x = 1 \\ 0 & \text{otherwise} \end{cases}$$

Requiring that f is *monotone* fixes this problem, i.e. $a \leq b \implies f(a) \leq f(b)$. Why?

Monotone isn't enough

Consider this function f defined over a cpo $(\mathbb{R} \cup \{-\infty, \infty\}, \leq)$:

$$f(x) = \begin{cases} \tan^{-1} x & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$$

Note that this function is not continuous at 0.

Oh no

It has a fixed point of 1, but the chain approaches 0:

$$\begin{aligned} f(-\infty) &= -\tfrac{\pi}{2} \\ f(-\tfrac{\pi}{2}) &= -1 \\ f(-1) &\approx -0.78 \end{aligned}$$

But $f(0) = 1$ — the least upper bound of the ascending Kleene chain is not the same as the least fixed point!

Continuity

Definition

In a cpo $(S, \sqsubseteq)$, a function $f : S \to S$ is *(Scott)-continuous* if, for every chain $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$, f preserves the least upper bound operator:

$$\bigsqcup_{n \in \mathbb{N}} f(x_n) = f\Big(\bigsqcup_{n \in \mathbb{N}} x_n\Big)$$

Theorem

Every Scott-continuous function is monotone. **Why?**

Requiring Scott-continuity instead of just monotonicity gives us the Kleene fixed point theorem...

The Kleene fixed point theorem

Theorem

Let $(S, \sqsubseteq)$ be a cpo and $f : S \to S$ be a Scott-continuous function. Then the lub of the Kleene ascending chain $\bigsqcup_{n\in\mathbb{N}} f^n(\bot)$ is the least fixed point of f.

Proof it is a fixed point:

$$
\begin{aligned}
f(\bigsqcup_{n\in\mathbb{N}} f^n(\bot)) &= \bigsqcup_{n\in\mathbb{N}} f(f^n(\bot)) && \text{(continuity)} \\
&= \bigsqcup_{n\in\mathbb{N}} f^{n+1}(\bot) \\
&= \bigsqcup_{n=1,2\ldots} f^n(\bot) && \text{(reindexing)} \\
&= \bot \sqcup \bigsqcup_{n=1,2\ldots} f^n(\bot) \\
&= \bigsqcup_{n\in\mathbb{N}} f^n(\bot)
\end{aligned}
$$

Proof of the FPT

Proof it is the least fixed point:

Let y be a fixed point of f. We know that $\perp \sqsubseteq y$ by definition of $\perp$. Taking f of both sides, we get $f(\perp) \sqsubseteq y$. We can continue this inductively and thus we know that, for all $n \in \mathbb{N}$, $f^n(\perp) \sqsubseteq y$. Because y is an upper bound of the Kleene ascending chain, it must also be at least as large as the lub of that chain.

Bringing it back to semantics

For our programming language, our cpo is $(\mathcal{P}(\Sigma \times \Sigma), \subseteq)$:

- The least element $\bot = \emptyset$
- The least upper bound of a chain $f_0 \subseteq f_1 \subseteq f2 \ldots$ is just $\bigcup_{i \in \mathbb{N}} f_i$

All of our composite operators are Scott-continuous:

$$[\![P + Q]\!] = [\![P]\!] \cup [\![Q]\!] \qquad [\![P; Q]\!] = [\![P]\!] \fatsemi [\![Q]\!]$$

Thus, we know from the fixed point theorem that least solutions to our recursive equations always exist and they can be found by iteratively applying the function until we find a fixed point.

Non-termination

Consider a program that may loop forever, such as $(x := x + 1)^\star$.

Problem

This possibility is not captured in our semantics!

Programs that definitely loop forever, like $(x := x + 1)^\star; x = 0$ have identical semantics to programs that always fail like $1 = 2$.

Key idea

Add a special value, confusingly also written $\perp$, which represents non-terminating computations. Our models would now be $\mathcal{P}(\Sigma \times \Sigma_\perp)$ where $\Sigma_\perp$ is either a state or the special “loop forever” value.

Representing non-termination

The "loop forever" value must show up in the least element of the cpo. Why?
If I have a recursive equation $[\![R]\!] = [\![R]\!]$, this ought to represent looping forever.

Problem

Our ordering says the model is "greater" when we remove $\bot$, but "smaller" when we remove anything else, and vice versa.

It's quite tricky to define this ordering such that it is a cpo and such that our language operations are still continuous.

Further reading

Plotkin resolved this with his Powerdomain construction, which gives a general treatment of non-determinism such that any cpo can be lifted to a non-deterministic context.

Common Theorems

It is typical to define both operational and denotational models for the same language and then prove theorems that relate them.

Definition

Let $(\sigma, P) \Downarrow \sigma'$ be an operational semantics for our language. It says that, starting in state σ, evaluating the program P on a machine results in σ'.

- *Soundness* If $(\sigma, P) \Downarrow \sigma'$ then $(\sigma, \sigma') \in [\![P]\!]$
- *Adequacy* If $(\sigma, \sigma') \in [\![P]\!]$ then $(\sigma, P) \Downarrow \sigma'$
- *Full Abstraction* $[\![P]\!] = [\![Q]\!]$ iff for all contexts C and states σ and σ', $(\sigma, C[P]) \Downarrow \sigma' \Leftrightarrow (\sigma, C[Q]) \Downarrow \sigma'$

The first two are common. The last one is hard.

More on denotations

This is just the tip of the iceberg in Denotational Semantics.

- Effectful programs use Kleisli categories (monads) for their domain
- Categorical semantics which use structures from category theory for denotations.
- Game semantics which use games as denotations.
- Probabilistic powerdomains and quasi-Borel spaces for probablistic programs.
- Concurrency semantics using traces, transition systems, event structures, Petri nets and so on. MCS

Farewell

Best of luck with your exams and the rest of your life! Please feel free to reach out if you’re interested in learning more theory.