Chapter 2: Denotational Semantics

Bohua Zhan

Institute of Software, Chinese Academy of Sciences

December 2023

Reading

- Concrete Semantics (Nipkow & Klein), Chapter 11
- Introduction to Formal Semantics (Zhou & Zhan), Chapter 2

Denotational Semantics

Two principles:

- Assign a *mathematical object* to each program (e.g. relation or partial function).
- The object assigned to a program should be defined in terms of objects assigned to its components.

IMP language

Recall the IMP language from the previous chapter:

$$
\begin{aligned}
com = {} & \mathbf{skip} && \text{(skip)} \\
\mid {} & var := aexp && \text{(assign)} \\
\mid {} & com;\ com && \text{(seq)} \\
\mid {} & \mathbf{if}\ bexp\ \mathbf{then}\ com\ \mathbf{else}\ com && \text{(if)} \\
\mid {} & \mathbf{while}\ bexp\ \mathbf{do}\ com && \text{(while)}
\end{aligned}
$$

where state is a function $var \to int$.

Denotation of expressions

We already have simple examples of denotations: each arithmetic or boolean expression can be considered as functions from states to numbers or boolean values.

- For an arithmetic expression e, its denotation $[\![e]\!]$ maps a state s to $[\![e]\!]_s$, so $[\![\cdot]\!]$ has type

  $$aexp \Rightarrow (state \Rightarrow int).$$

  Compositional since $[\![e_1 + e_2]\!]_s = [\![e_1]\!]_s + [\![e_2]\!]_s$.
- For a boolean expression b, its denotation $[\![b]\!]$ maps a state s to $[\![b]\!]_s$, so $[\![\cdot]\!]$ has type

  $$bexp \Rightarrow (state \Rightarrow bool).$$

  Compositional since $[\![e_1 < e_2]\!]_s = [\![e_1]\!]_s < [\![e_2]\!]_s$.

Denotation of programs

Define denotation as mapping from programs to relations on states:

$$D :: com \Rightarrow (state \times state)\ set.$$

where $(s, t) \in D(c)$ means c carries state s to state t.

Can't we just define $(s, t) \in D(c)$ if and only if $(c, s) \Rightarrow t$?

Does not satisfy the second principle: object assigned to $\mathbf{while}\ b\ \mathbf{do}\ c$ is not defined in terms of object assigned to c.

Basic rules

We start from the beginning. Rules for the basic commands are:

$$
\begin{aligned}
&D(\mathbf{skip}) = Id \\
&D(v := e) = \{(s,t).\ t = s(x := [\![e]\!]_s)\} \\
&D(c_1; c_2) = D(c_1) \circ D(c_2) \\
&D(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) = \\
&\qquad \{(s,t).\ \text{if } [\![b]\!]_s \text{ then } (s,t) \in D(c_1) \text{ else } (s,t) \in D(c_2)\}
\end{aligned}
$$

here $\cdot \circ \cdot$ denotes composition of two relations.

Denotation for **while**

$$D(\textbf{while } b \textbf{ do } c) = ???$$

Definition requires use of a *fixed point* (or fixpoint).

Intuition: let $w = \textbf{while } b \textbf{ do } c$. Then $D(w)$ should contain all (s, s) where $[\![b]\!]_s = \text{false}$, and all $(s, t) \in D(c) \circ D(w)$ where $[\![b]\!]_s = \text{true}$. That is, define $W_{b,c}$ by:

$$W_{b,c}(S) = \{(s,t).\ \text{if } [\![b]\!]_s \text{ then } (s,t) \in D(c) \circ S \text{ else } s = t\},$$

then $W_{b,c}(D(w)) \subseteq D(w)$.

Denotation for **while**

Note $W_{b,c}$ is a monotonic function on relations, meaning

$$S \subseteq T \Longrightarrow W_{b,c}(S) \subseteq W_{b,c}(T).$$

Hence,

$$\emptyset \subseteq W_{b,c}(\emptyset) \subseteq W_{b,c}^2(\emptyset) \subseteq W_{b,c}^3(\emptyset) \subseteq \cdots$$

We may hope $W_{b,c}^n(\emptyset)$ converges to a limit, and this is indeed the case.

Define $W_{b,c}^\infty(\emptyset)$ as the limit, and define:

$$D(\textbf{while } b \textbf{ do } c) = W_{b,c}^\infty(\emptyset)$$

Example

How do we understand the relations $W_{b,c}^i(\emptyset)$?

$$\begin{aligned} W_{b,c}(\emptyset) &= \{(s,t).\ \text{if } [\![b]\!]_s \text{ then } (s,t) \in D(c) \circ \emptyset \text{ else } s = t\}\} \\ &= \{(s,t).\ \neg[\![b]\!]_s \wedge s = t\} \end{aligned}$$

That is, pairs of starting/ending states corresponding to immediate exit of the loop.

$$\begin{aligned} W_{b,c}^2(\emptyset) &= \{(s,t).\ \text{if } [\![b]\!]_s \text{ then } (s,t) \in D(c) \circ W_{b,c}(\emptyset) \text{ else } s = t\}\} \\ &= \{(s,t).\ [\![b]\!]_s \wedge (s,t) \in D(c) \circ W_{b,c}(\emptyset)\} \\ &\qquad \cup \{(s,t).\ \neg[\![b]\!]_s \wedge s = t\} \end{aligned}$$

That is, pairs of starting/ending states corresponding to one or zero iterations through the loop.

Observations

- In general, $W_{b,c}^{n+1}(\emptyset)$ is the relation given by allowing at most n iterations of the loop.
- $W_{b,c}^{\infty}(\emptyset)$ is a fixpoint of $W_{b,c}$, that is

$$W_{b,c}(W_{b,c}^{\infty}(\emptyset)) = W_{b,c}^{\infty}(\emptyset)$$

- It may be theoretically cleaner to define D directly as the least fixpoint of $W_{b,c}$:

$$D(\textbf{while } b \textbf{ do } c) = \mathit{lfp}(W_{b,c})$$

Least fixpoint

Why define as the least fixpoint?

Consider the simplest infinite loop program:

$$\textbf{while } \mathit{true} \textbf{ do skip}$$

The denotation $D(\textbf{skip})$ is Id, so

$$\begin{aligned} W_{b,\textbf{skip}}(S) &= \{(s,t).\ \text{if } \mathit{true} \text{ then } (s,t) \in D(\textbf{skip}) \circ S \text{ else } s = t\} \\ &= \{(s,t).\ (s,t) \in D(\textbf{skip}) \circ S\} \\ &= S \end{aligned}$$

In other words, any set S is a fixpoint of $W_{b,\textbf{skip}}$. Taking the least fixpoint agrees with our definition that there is no t such that $(c,s) \Rightarrow t$ if c is non-terminating on s.

Does the least fixpoint exist?

Yes, by the **Knaster-Tarski fixpoint theorem**:

定理 (Knaster-Tarski, for functions on sets)

If f is a monotone function, then the least fixpoint exists and can be obtained by:

$$lfp(f) = \bigcap\{P. f(P) \subseteq P\}$$

We say P is a pre-fixpoint of f if $f(P) \subseteq P$, then the above theorem states that the least fixpoint of f is the intersection of all pre-fixpoint of f.

Equivalence of denotational and big-step semantics

The equivalence between denotational semantics and big-step semantics is given by the following theorem:

定理 (Equivalence of denotational and big-step semantics)

$(s, t) \in D(c) \longleftrightarrow (c, s) \Rightarrow t$

Proof (sketch)

Only the *WhileTrue* case is nontrivial. To show

$$(\mathbf{while}\ b\ \mathbf{do}\ c, s) \Rightarrow t \longrightarrow (s, t) \in D(\mathbf{while}\ b\ \mathbf{do}\ c),$$

we proceed by induction on the proof of $(c, s) \Rightarrow t$. In the *WhileTrue* case, we have $[\![b]\!]_s = \text{true}$, $(s, s') \in c$, $(s', t) \in D(\mathbf{while}\ b\ \mathbf{do}\ c)$. Then the conclusion follows from the fact that

$$W_{b,c}(D(\mathbf{while}\ b\ \mathbf{do}\ c)) \subseteq D(\mathbf{while}\ b\ \mathbf{do}\ c).$$

Equivalence of denotational and big-step semantics

Proof (sketch)

Now consider the other direction, showing

$$(s, t) \in D(\textbf{while } b \textbf{ do } c) \longrightarrow (\textbf{while } b \textbf{ do } c, s) \Rightarrow t$$

The idea is to show that the relation

$$B(b, c) = \{(s, t).\ (\textbf{while } b \textbf{ do } c, s) \Rightarrow t\}$$

is a pre-fixpoint of $W_{b,c}$. Since $D(\textbf{while } b \textbf{ do } c)$ is defined as the *least* fixpoint, it is a subset of $B(b, c)$, hence the implication follows.

Equivalence of programs

A nice corollary is that, two commands are *equivalent* according to big-step semantics if and only if they have the same denotation:

推论

$c_1 \sim c_2 \longleftrightarrow D(c_1) = D(c_2)$

Remarks

We have only seen the tip of the iceberg of a sophisticated theory, originating from the work of Dana Scott and Christopher Strachey. Further study will involve concepts like:

- Lattices and complete lattices.
- Continuous functions on lattices.
- Kleene fixpoint theorem and Knaster-Tarski fixpoint theorem.
- and so on. . .

Consult the references if you are interested in learning more.

Summary

We have studied three ways to assign meaning to programs:

- **Big-step operational semantics:** statements of the form $(c, s) \Rightarrow t$, takes all steps of execution at once.
- **Small-step operational semantics:** statements of the form $(c, s) \rightarrow (c', t)$, takes steps one-at-a-time. Infinite chains to indicate non-termination.
- **Denotational semantics:** assign mathematical objects (relations or partial functions) to programs. Compositional definitions.

Having defined what a program does, we next consider methods for proving that a program is correct.