Styles of formal semantics

Operational: Meanings for program phrases defined in terms of the steps of computation they can take during program execution.

Axiomatic: Meanings for program phrases defined indirectly via the axioms and rules of some logic of program properties (Hoare logic).

Denotational: Concerned with giving mathematical models of programming languages. Meaning for program phrases defined abstractly as elements of suitable mathematical structure.
Operational vs denotational semantics

Operational semantics defines an interpreter, from abstract syntax to abstract syntax. Metalanguage is inference rules (slides) or OCaml (interp.ml)

Denotational semantics defines a compiler (translator), from abstract syntax to a different language with known semantics

Target language is math, but we’ll make it a tiny core of OCaml (hence “pseudo”)

Metalanguage is math or OCaml (we’ll show both)
Operational vs denotational semantics (cont.)

Operational semantics is:

- simple
- of many flavors (natural/large-step, small-step, more or less abstract)
- good for: language definition, verifying language properties (e.g., IMP is deterministic), verifying correctness of tools that manipulate programs (e.g., interpreters, compilers, type checkers, etc.)
- disadvantages: awkward for verifying even simple programs (remember lecture 3?), not compositional

Denotational semantics is:

- mathematical (the meaning of a syntactic expression is a mathematical object)
- compositional

Denotational semantics is also called: fixed-point semantics, mathematical semantics, Scott-Strachey semantics.
Denotational semantics

Each phrase (i.e., part of a program), \( P \), is given a *denotation*, \([ [P] ]\) – a mathematical object representing the contribution of \( P \) to the meaning of *any* complete program in which it occurs.

The denotation of a phrase is determined just by the denotations of its sub phrases (i.e., the semantics is *compositional*).

What does it mean for a formal semantic system to be compositional? It means that the collection of mathematical objects we use to give denotations to program phases has to be sufficiently rich that it supports operations for modeling all the phrase-forming constructs of the program language in question (e.g., while loops can be tricky; more on that later).
The basic idea

A heap is a math/ML function from strings to integers:

\[ \text{string} \rightarrow \text{int} \]

An expression denotes a math/ML function from heaps to integers

\[ \text{den}(e) : (\text{string} \rightarrow \text{int}) \rightarrow \text{int} \]

A statement denotes a math/ML function from heaps to heaps

\[ \text{den}(s) : (\text{string} \rightarrow \text{int}) \rightarrow (\text{string} \rightarrow \text{int}) \]

Now just define \( \text{den} \) in our metalanguage (math or ML), inductively over the source language abstract syntax
Expressions

\[ \text{den}(e) : (\text{string} \rightarrow \text{int}) \rightarrow \text{int} \]

\[
\begin{align*}
\text{den}(c) & \quad = \quad \text{fun } h \rightarrow c \\
\text{den}(x) & \quad = \quad \text{fun } h \rightarrow h \ x \\
\text{den}(e_1 + e_2) & \quad = \quad \text{fun } h \rightarrow (\text{den}(e_1) \ h) + (\text{den}(e_2) \ h) \\
\text{den}(e_1 \times e_2) & \quad = \quad \text{fun } h \rightarrow (\text{den}(e_1) \ h) \times (\text{den}(e_2) \ h)
\end{align*}
\]

In plus (and times) case, two “ambiguities”:

- “+” from meta language or target language?
  - Translate abstract + to OCaml +, (ignoring overflow)

- \(\text{When do we denote } e_1 \text{ and } e_2?\)
  - Not a focus of the metalanguage. At “compile time”. 
Switching metalanguage

With OCaml as our metalanguage, ambiguities go away

But it is harder to distinguish mentally between “target” and “meta”

If denote in function body, then source is “around at run time”

- After translation, should be able to “remove” the definition of the abstract syntax
- ML does not have such a feature, but the point is we no longer need the abstract syntax

See denote.ml
Statements, w/o while

\[ den(s) : (\text{string} \rightarrow \text{int}) \rightarrow (\text{string} \rightarrow \text{int}) \]

\[
\begin{align*}
\text{den(skip)} & = \text{fun } h \rightarrow h \\
\text{den}(x := e) & = \text{fun } h \rightarrow (\text{fun } v \rightarrow \text{if } x = v \text{ then } \text{den}(e) \text{ h else } h \text{ v}) \\
\text{den}(s_1; s_2) & = \text{fun } h \rightarrow \text{den}(s_2) (\text{den}(s_1) \text{ h}) \\
\text{den}(\text{if } e \ s_1 \ s_2) & = \text{fun } h \rightarrow \text{if } \text{den}(e) \text{ h} > 0 \text{ then } \text{den}(s_1) \text{ h else } \text{den}(s_2) \text{ h}
\end{align*}
\]

Same ambiguities; same answers

See denote.ml
The function denoting a while statement is inherently recursive!

Good thing our target language has recursive functions!

Why doesn’t $\text{den}(\text{while } e \ s) = \text{den}(\text{if } e \ (s; \text{while } e \ s) \ \text{skip})$ make any sense?
Two common mistakes

A denotational semantics should “eagerly” translate the entire program

- E.g., both branches of an if

But a denotational semantics should “terminate”

- I.e., avoid any circular definitions \textit{in the translating}
- The \textit{result} of the translation can use (well-founded) recursion
- E.g., compiling a while-loop should not produce an infinite amount of code
Finishing the story

```ocaml
let denote_prog s =
    let d = denote_stmt s in
    fun () -> (d (fun x -> 0)) "ans"
```

Compile-time: let x = denote_prog (parse file)

Run-time: print_int (x ())

In-between: We have a OCaml program using only functions, variables, ifs, constants, +, *, >, etc.

▶ Does not use any constructors of exp or stmt (e.g., Seq)
The real story

For “real” denotational semantics, target language is math

(And we write $[s]$ instead of $\text{den}(s)$)

Example: $[x := e][H] = [H][x \mapsto [e][H]]$

There are two major problems, both due to while:

1. Math functions do not diverge, so no function denotes
   \textbf{while 1 skip}

2. The denotation of loops cannot be circular
The elevator version, which we will not pursue

For (1), we “lift” the semantic domains to include a special \( \bot \)

\[
\text{den}(s) : (\text{string} \to \text{int}) \to ((\text{string} \to \text{int}) \cup \bot)
\]

- Have to change meaning of \( \text{den}(s_2) \circ \text{den}(s_1) \) appropriately

For (2), we use \textbf{while} \( e \ s \) to define a (meta)function \( f \) that given a lifted heap-transformer \( X \) produces a lifted heap-transformer \( X' \):

- If \( \text{den}(e)(\text{den}(H)) = 0 \), then \( \text{den}(H) \)
- Else \( X \circ \text{den}(s) \)

Now let \( \text{den}(\textbf{while} \ e \ s) \) be the least fixed-point of \( f \)

- An hour of math to prove the least fixed-point exists
- Another hour to prove it is the limit of starting with \( \bot \) and applying \( f \) over and over (i.e., any number of loop iterations)
- Keywords: monotonic functions, complete partial orders, Knaster-Tarski theorem
Practical applications

Denotational semantics is used in practice as the basis for static analysis of programs (all compilers and many other tools do this!)

- Used for discovering properties of all possible dynamic executions (e.g., for code optimization, bug finding, security, ...)

Active area of research, key conferences:

- Principles of Programming Languages (POPL), Computer-Aided Verification (CAV), Programming Language Design and Implementation (PLDI).

CIS 410/510 Program Analysis and Transformation covers some of this theory and applies it to real software.)
Where we are

- Have seen operational and denotational semantics
- Connection to interpreters and compilers
- Useful for rigorous definitions and proving properties
- Next: Equivalence of semantics
  - Crucial for compiler writers
  - Crucial for code maintainers
- Then: Leave IMP behind and consider functions

But first: Will any of this help write an O/S service?