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)

The basic idea

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

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

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

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

- \[ \text{den}(c) = \text{fun } h \rightarrow c \]
- \[ \text{den}(x) = \text{fun } h \rightarrow h \times \]
- \[ \text{den}(e_1 + e_2) = \text{fun } h \rightarrow (\text{den}(e_1) h) + (\text{den}(e_2) h) \]
- \[ \text{den}(e_1 \times e_2) = \text{fun } h \rightarrow (\text{den}(e_1) h) \times (\text{den}(e_2) h) \]

In plus (and times) case, two "ambiguities":
- "+" from meta language or target language?
  - Translate abstract + to OCaml +, (ignoring overflow)
- When do we denote \( e_1 \) and \( e_2 \)?
  - Not a focus of the metalanguage. At "compile time".

\[ \text{den}(\text{skip}) = \text{fun } h \rightarrow h \]
\[ \text{den}(x := e) = \text{fun } h \rightarrow (\text{fun } v \rightarrow \text{if } x = v \text{ then } \text{den}(e) h \text{ else } h \cdot v) \]
\[ \text{den}(\text{if } e \cdot s_1 ; s_2) = \text{fun } h \rightarrow \text{den}(s_2) \cdot (\text{den}(s_1) h) \]
\[ \text{den}(\text{while } e \cdot s) = \text{fun } h \rightarrow \text{if } (\text{den}(e) h) > 0 \text{ then } \text{den}(s_1) h \text{ else } \text{den}(s_2) h \]

Same ambiguities; same answers

See denote.ml

### 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

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

- \[ \text{den}(\text{skip}) = \text{fun } h \rightarrow h \]
- \[ \text{den}(x := e) = \text{fun } h \rightarrow (\text{fun } v \rightarrow \text{if } x = v \text{ then } \text{den}(e) h \text{ else } h \cdot v) \]
- \[ \text{den}(\text{if } e \cdot s_1 ; s_2) = \text{fun } h \rightarrow \text{den}(s_2) \cdot (\text{den}(s_1) h) \]

The function denoting a while statement is inherently recursive!

Good thing our target language has recursive functions!

Why doesn’t \( \text{den}((\text{while } e \cdot s) \cdot \text{skip}) \) 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 in the translating
- The result of the translation can use (well-founded) recursion
- E.g., compiling a while-loop should not produce an infinite amount of code

### While

\[ \text{den}(\text{while } e \cdot s) = \text{fun } h \rightarrow \text{if } (\text{den}(e) h) > 0 \text{ then } \text{den}(s_1) h \text{ else } \text{den}(s_2) h \]

The function denoting a while statement is inherently recursive!

Good thing our target language has recursive functions!

Why doesn’t \( \text{den}((\text{while } e \cdot s) \cdot \text{skip}) \) make any sense?

### Finishing the story

\[ \text{let } \text{denote}_\text{prog} \text{ s } = \]
\[ \text{let } d = \text{denote}_\text{stmt} \text{ s } \text{ in} \]
\[ \text{fun } () \rightarrow (d \cdot (\text{fun } x \rightarrow 0)) \text{ "ans"} \]

Compile-time: let \( x = \text{denote}_\text{prog} (\text{parse file}) \)

Run-time: \( \text{print_int} (x ()) \)

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 $\llbracket s \rrbracket$ instead of $\text{den}(s)$)

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

There are two major problems, both due to while:
1. Math functions do not diverge, so no function denotes while 1 skip
2. The denotation of loops cannot be circular

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?