A different approach

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:

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

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

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

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

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

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

▶ When do we denote \( e_1 \) 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

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

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

Same ambiguities; same answers

See denote.ml
While

\[
\text{den(while } e \text{ s) = while(e,s) -> let rec } f \ h = \begin{cases} \text{if (den(e) } h) > 0 & \text{let den } f \ e \ \text{in} \\ \text{then } f \ (\text{den(s) } h) & \text{let rec } f \ h = \\ \text{else } h \ \text{in} & \text{if (d1 } h) > 0 \\ & \text{then } f \ (d2 } h) \\ & \text{else } h \ \text{in} \\ & f \end{cases}
\]

The function denoting a while statement is inherently recursive!

Good thing our target language has recursive functions!

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

The real story

For “real” denotational semantics, target language is math

\( (And \ we \ write \ [s] \ instead \ of \ den(s)) \)

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 \( \text{while } 1 \text{ skip} \)
2. The denotation of loops cannot be circular

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?