CSE 505: Concepts of Programming Languages

Dan Grossman
Fall 2007
Lecture 4—“Denotational” Semantics for IMP
Today’s Plan

- Finish proofs and motivation from last time
- “Denotational” semantics via translation to ML

Goal: Saying “Let’s consider the trade-offs of using a denotational semantics to achieve a high-performance, extensible operating system” with a straight face.
Example 1 *summary*

Theorem: If $\text{noneg}(H)$, $\text{noneg}(s)$, and $H \; s \rightarrow^n H' \; s'$, then $\text{noneg}(H')$ and $\text{noneg}(s')$.

Proof: By induction on $n$. $n = 0$ is immediate. For $n > 0$, use lemma: If $\text{noneg}(H)$, $\text{noneg}(s)$, and $H \; s \rightarrow H' \; s'$, then $\text{noneg}(H')$ and $\text{noneg}(s')$.

Proof: By induction on derivation of $H \; s \rightarrow H' \; s'$. Consider bottom-most (last) rule used: Cases Seq1, If1, If2, and While straightforward.

Case Seq2 uses induction ($s = s_1; s_2$ and $H \; s_1 \rightarrow H' \; s'_1$ via a shorter derivation).
Example 1 cont’d

Case Assign uses a lemma: If $\textit{noneg}(H)$, $\textit{noneg}(e)$, and $H \; ; e \downarrow c$, then $\textit{noneg}(c)$. Proof: Induction on derivation. Plus and Times cases use induction and math facts.

Motivation: We preserved a nontrivial property of our program state. It would fail if we had

- Overly flexible rules, e.g.:

  $$ H \; ; c \downarrow c' $$

- An “unsafe” language like C:

  $$ H(x) = \{c_0, \ldots, c_{n-1}\} \quad H \; ; e \downarrow c \quad c \geq n $$

  $$ H \; ; x[e] := e' \rightarrow H' \; ; s' $$
Example 2

Theorem: If for all $H$, we know $s_1$ and $s_2$ terminate, then for all $H$, we know $H; (s_1; s_2)$ terminates.

Seq Lemma: If $H; s_1 \rightarrow^n H'; s_1'$, then $H; s_1; s_2 \rightarrow^n H'; s_1'; s_2$. Proof: Induction on $n$.

Using lemma, theorem holds in $n + 1 + m$ steps where $H; s_1 \rightarrow^n H'; \text{skip}$ and $H'; s_2 \rightarrow^m H''; \text{skip}$.

Motivation: Termination is often desirable. Can sometimes prove it for a sublanguage (e.g., while-free IMP programs) or for “YVIP”.
Even more general proofs to come

We defined the semantics.

Given our semantics, we established properties of programs and sets of programs.

More interesting is having multiple semantics—for what program states are they equivalent? (For what notion of equivalence?)

Or having a more abstract semantics (e.g., a type system) and asking if it is preserved under evaluation. (If $e$ has type $\tau$ and $e$ becomes $e'$, does $e'$ have type $\tau$?)

But first a one-lecture detour to “denotational” semantics.
A different approach

Operational semantics defines an interpreter, from abstract syntax to abstract syntax. Metalanguage is inference rules (slides) or Caml (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 Caml for now.

Metalanguage is math or Caml (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.
### Expressions

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

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

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

- "+" from source language or target language?
  - Translate abstract + to Caml +, ignoring overflow (!)

- *when* do we denote \( e_1 \) and \( e_2 \)?
  - Not a focus of the metalanguage. At “compile time”.

Dan Grossman  
CSE505 Fall 2007, Lecture 4  
9
Switching metalanguage

With Caml as our metalanguage, ambiguities go away.

But it’s harder to distinguish mentally between “target” and “meta”.

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

(See denote.ml.)
Statements, w/o while

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

\[\text{den(\text{skip})} = \text{fun} \ h \rightarrow h\]
\[\text{den(\text{x := e})} = \text{fun} \ h \rightarrow (\text{fun} \ v \rightarrow \text{if} \ x=v \ \text{then} \ \text{den(e) \ h \ else} \ h \ v)\]
\[\text{den(\text{s1; s2})} = \text{fun} \ h \rightarrow \text{den(s2) \ (den(s1) \ h)}\]
\[\text{den(\text{if e s1 s2})} = \text{fun} \ h \rightarrow \text{if} \ \text{den(e) \ h} > 0 \ \text{then} \ \text{den(s1) \ h} \ \text{else} \ \text{den(s2) \ h}\]

Same ambiguities; same answers.

See denote.ml.
While

\[ \text{den} (\text{while } e \ s) = \quad | \ \text{While}(e, s) \rightarrow \]

let rec \( f \ h = \) let \( d1 = \text{denote}_{\text{exp}} \ e \) in
  \[ \text{if } \left( \text{den}(e) \ h \right) > 0 \quad \text{let } d2 = \text{denote}_{\text{stmt}} \ s \text{ in} \]
  \[ \text{then } f \ (\text{den}(s) \ h) \quad \text{let } \text{rec } f \ h = \]
  \[ \text{else } h \text{ in} \quad \text{if } (d1 \ h) > 0 \]
  \[ f \quad \text{then } f \ (d2 \ h) \]
  \[ \text{else } h \text{ in} \]
  \[ f \]

The function denoting a while statement is inherently recursive!

Good thing our target language has recursive functions!
Finishing the story

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 Caml program, so many tools available, but target language should be a good match.
The real story

For “real” denotational semantics, target language is math (And we write \([s]\) instead of \(\text{den}(s)\))

Example: \([x := e][H] = [H][x \mapsto [e]]\)

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

For (1), we “lift” the semantic domains to include a special \( \bot \). (So 
\( \text{den}(s) : \{ \bot, \text{string} \rightarrow \text{int} \} \rightarrow \{ \bot, \text{string} \rightarrow \text{int} \} \).)

For (2), we define a (meta)function \( f \) to generate a sequence of 
denotations: “\( \bot \)”, “\( \leq 1 \text{ iteration then } \bot \)”, “\( \leq 2 \text{ iterations then } \bot \)”, and we 
denote the loop via the least fixed point of \( f \). (Intuitively, a countably 
infinite number of iterations.)

Proving this fixed point is well-defined takes a lecture of math (keywords: 
monotonic functions, complete partial orders, Knaster-Tarski theorem)

I promise not to say those words again in class.

You promise not to take this description too seriously.
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?