CSE505: Graduate Programming Languages

Lecture 5 — Pseudo-Denotational Semantics

Dan Grossman
Winter 2012
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 a tiny core of Caml (hence “pseudo”).

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

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

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

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

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

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

- After translation, should be able to “remove” the definition of the abstract syntax

- ML doesn’t have such a feature, but the point is we no longer need the abstract syntax

See denote.ml
Statements, w/o while

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

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

Same ambiguities; same answers

See denote.ml
While

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

The function denoting a while statement is inherently recursive!

Good thing our target language has recursive functions!

Why doesn’t \[ den(\textbf{while } e \ s) = den(\textbf{if } e \ (s; \textbf{while } e \ s) \ \textbf{skip}) \]
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
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 using only functions, variables, ifs, constants, +, *, >, etc.
  ▶ Doesn’t 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 \textit{major} problems, both due to \textit{while}:

1. Math functions do not diverge, so no function denotes \textit{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 $\perp$

$\text{den}(s) : (\text{string} \rightarrow \text{int}) \rightarrow ((\text{string} \rightarrow \text{int}) \cup \perp)$

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

For (2), we use $\text{while } e \ s$ 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 $\text{den}(s) \circ X$

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

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