CSE 505: Concepts of Programming Languages

Dan Grossman
Fall 2003
Lecture 4— “Denotational” Semantics for IMP

(Bonus: Connection to reality via packet filters)
Today’s Plan

- Finish example proofs
- Motivate doing this “obvious” stuff via “wrong” rules
- “Denotational” semantics via translation to ML
- Real-world example: packet filters

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 $\text{nong}(H)$, $\text{nong}(s)$, and $H ; s \rightarrow^n H' ; s'$, then $\text{nong}(H')$ and $\text{nong}(s')$.

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

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 \( \text{nong}(H) \), \( \text{nong}(e) \), and \( H \; ; \; e \Downarrow c \), then \( \text{nong}(c) \). 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 $H \trianglerighteq s_1 \rightarrow^n H' \trianglerighteq s_1'$, then $H \trianglerighteq s_1; s_2 \rightarrow^n H' \trianglerighteq s_1'; s_2$. Proof: Induction on $n$.

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

Motivation: Termination is often desirable. Can sometimes prove it for a sublanguage (e.g., while-free IMP programs) or for “YVIP”.
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 OCaml for now. Metalanguage is math or OCaml (we’ll show both).
The basic idea

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

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

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

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

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

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

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

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

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

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

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

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

Dan Grossman  

CSE505 Fall 2003, Lecture 4
Switching metalanguage

\[
\begin{align*}
\text{let rec denote_exp } & \text{ e } = \text{ match e with} \\
\text{   Int } i & \text{ -> (fun } h \text{ -> } i) \\
| \text{ Var } v & \text{ -> (fun } h \text{ -> } h \ v) \\
| \text{ Plus(e1,e2)} & \text{ ->} \\
\text{   let } d1 = \text{ denote_exp } e1 & \text{ in} \\
\text{   let } d2 = \text{ denote_exp } e2 & \text{ in} \\
\text{   (fun } h \text{ -> (d1 } h \text{) + (d2 } h\text{))} \\
| \text{ Times(e1,e2)} & \text{ ->} \\
\text{   let } d1 = \text{ denote_exp } e1 & \text{ in} \\
\text{   let } d2 = \text{ denote_exp } e2 & \text{ in} \\
\text{   (fun } h \text{ -> (d1 } h \text{) * (d2 } h\text{))}
\end{align*}
\]

Ambiguities go away, but meta and target language the same.

If denote in function body, then source is “around at run time”.
Statements, w/o while

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

\[
den(\text{skip}) = \text{fun } h \rightarrow h
\]

\[
den(x := e) = \text{fun } h \rightarrow (\text{fun } v \rightarrow \text{if } x = v \text{ then } \text{den}(e) h \text{ else } h v)
\]

\[
den(s_1; s_2) = \text{fun } h \rightarrow \text{den}(s_2) (\text{den}(s_1) h)
\]

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

Same ambiguities; same answers.
let rec denote_stmt s = match s with
    Skip -> (fun h -> h)
| Assign(v,e) ->
    let d = denote_exp e in
    (fun h ->
    let c = d h in
    fun x -> if x=v then c else h x)
(* ....... omitting Seq ....... *)
| If(e,s1,s2) ->
    let d1 = denote_exp e in
    let d2 = denote_stmt s1 in
    let d3 = denote_stmt s2 in
    (fun h -> if (d1 h)>0 then (d2 h) else (d3 h))
While

\[
\text{\texttt{den(while } e \texttt{ s) = \text{\texttt{While(e,s) \rightarrow}}} \\
\text{let rec } f \texttt{ h =} \\
\quad \text{let d1=denote\_exp } e \texttt{ in} \\
\quad \text{if (\texttt{den(e) h})>0} \\
\quad \text{let d2=denote\_stmt } s \texttt{ in} \\
\quad \text{then } f \texttt{ (den(s) h) \text{\texttt{}}} \\
\quad \text{else } h \texttt{ in} \\
\quad f
\]

The function denoting a while statement is inherently recursive!

Good thing our target language has recursive functions!
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 an OCaml 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 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 $\text{den}(s) : \{ \bot, \text{string} \to \text{int} \} \to \{ \bot, \text{string} \to \text{int} \}$.)

For (2), we define a (meta)function $f$ to generate a sequence of denotations: “$\bot$”, “$\leq 1$ iteration then $\bot$”, “$\leq 2$ 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?
Packet Filters

Almost everything I know about packet filters:

- Some bits come in off the wire
- Some application(s) want the “packet” and some do not (e.g., port number)
- For safety, only the O/S can access the wire.
- For extensibility, only an application can accept/reject a packet.

Conventional solution goes to user-space for every packet and app that wants (any) packets.

Faster solution: Run app-written filters in kernel-space.
What we need

Now the O/S writer is defining the packet-filter language!

Properties we wish of (untrusted) filters:

1. Don’t corrupt kernel data structures
2. Terminate (within a time bound)
3. Run fast (the whole point)

Should we download some C/assembly code? (Get 1 of 3.)

Should we make up a language and “hope” it has these properties?
Language-based approaches

1. Interpret a language.
   + clean operational semantics, + portable, - may be slow (+ filter-specific optimizations), - unusual interface

2. Translate a language into C/assembly.
   + clean denotational semantics, + employ existing optimizers, - upfront cost, - unusual interface

3. Require a conservative subset of C/assembly.
   + normal interface, - too conservative w/o help

IMP has taught us about (1) and (2) — we’ll get to (3)