CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2005

Lecture 4— “Denotational” Semantics for IMP

(Bonus: Connection to reality via packet filters)

Today's Plan

- Finish proofs and motivation from last time
- "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 $noneg(H)$, $noneg(s)$, and $H\ ;\ s \rightarrow^n H'\ ;\ s'$, then $noneg(H')$ and $noneg(s')$.

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

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 $noneg(H)$, $noneg(e)$, and $H\ ;\ e \Downarrow c$, then $noneg(c)$. 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.:

$$\frac{}{H\ ;\ c \Downarrow c'}$$

- An "unsafe" language like C:

$$\frac{H(x) = \{c_0, \ldots, c_{n-1}\} \qquad H\ ;\ e \Downarrow c \qquad c \geq n}{H\ ;\ x[e] := e' \rightarrow H'\ ;\ s'}$$

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 $H \; ; \; s_1 \;\; \rightarrow^n \;\; H' \; ; \;$ **skip** and $H' \; ; \; s_2 \;\; \rightarrow^m \;\; H'' \; ; \;$ **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 τ and e becomes e', does e' have type τ?)

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 (translater), 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: $\boldsymbol{string} \rightarrow \boldsymbol{int}$

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

$$\boldsymbol{den(e) : (string \rightarrow int) \rightarrow int}$$

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

$$\boldsymbol{den(s) : (string \rightarrow int) \rightarrow (string \rightarrow int)}$$

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

Expressions

$$den(e) : (string \to int) \to int$$

$$\begin{aligned}
den(c) &= \texttt{fun h -> c} \\
den(x) &= \texttt{fun h -> h } x \\
den(e_1 + e_2) &= \texttt{fun h -> (}den(e_1)\texttt{ h) + (}den(e_2)\texttt{ h)} \\
den(e_1 * e_2) &= \texttt{fun h -> (}den(e_1)\texttt{ h) * (}den(e_2)\texttt{ h)}
\end{aligned}$$

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".

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

$$(string \to int) \to (string \to int)$$

$den(\textsf{skip}) \quad = \quad$ `fun h -> h`

$den(x := e) \quad =$

`fun h -> (fun v -> if` x`=v then` $den(e)$ `h else h v)`

$den(s_1; s_2) \quad = \quad$ `fun h ->` $den(s_2)$ `(`$den(s_1)$ `h)`

$den(\textsf{if}\ e\ s_1\ s_2) \quad =$

`fun h ->`

` if` $den(e)$ `h > 0 then` $den(s_1)$ `h else` $den(s_2)$ `h`

Same ambiguities; same answers.

See `denote.ml`.

While

$\boldsymbol{den}(\textbf{while}\ \boldsymbol{e}\ \boldsymbol{s}) =$

```
let rec f h =
     if (den(e) h)>0
     then f (den(s) h)
     else h in
 f
```

```
| While(e,s) ->
 let d1=denote_exp e in
 let d2=denote_stmt s in
 let rec f h =
   if (d1 h)>0
   then f (d2 h)
   else h 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 $den(s)$)

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 $den(s) : \{\bot, string \rightarrow int\} \rightarrow \{\bot, string \rightarrow int\}$.

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)