CSE 505: Concepts of Programming Languages

Dan Grossman

Fall 2009

Lecture 4— Proofs about Operational Semantics; Pseudo-Denotational Semantics

This lecture

- Continue last lecture, including detailed proofs (and some wrong approaches) of two "theorems"
  - How to prove them
  - Why these theorems are "interesting"
- The idea behind "denotational semantics" via translation to ML

Proofs

Write out proofs by hand for:

- **while** 1 **skip** diverges
  - Key point: Must get induction hypothesis "just right" — not too strong (false) or too weak (proof doesn't go through)
- "No negatives" is preserved by evaluation
  - Can define a program property via judgments and prove it holds after every step
  - "Inverting assumed derivations" gives you necessary facts for smaller expressions/statements (e.g., the while case)

Motivation of non-negatives

While "no negatives preserved" boils to down to properties of blue-+ and blue-*, writing out the whole proof ensures our language has no mistakes or bad interactions.

The theorem is false if we have:

- Overly flexible rules, e.g.:

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

- An "unsafe" language like C:

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

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: $\boldsymbol{string \rightarrow int}$

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

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

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

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".

- 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

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

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

$den(x := e)$ =

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

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

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

`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

den(**while** e 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 using only functions, variables, ifs, constants, +, *, >, etc.

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 **while** 1 **skip**.
2. The denotation of loops cannot be circular.

The elevator version; ignorable for 505

For (1), we "lift" the *semantic domains* to include a special $\perp$.
$\boldsymbol{den(s) : (string \rightarrow int) \rightarrow ((string \rightarrow int) \cup \perp)}$.

- Have to change meaning of $\boldsymbol{den(s_2) \circ den(s1)}$ appropriately.

For (2), we use **while** $\boldsymbol{e\ s}$ to define a (meta)function $\boldsymbol{f}$ that given a lifted heap-transformer $\boldsymbol{X}$ produces a lifted heap-transformer $\boldsymbol{X'}$:

- If $\boldsymbol{den(e)(den(H)) = 0}$, then $\boldsymbol{den(H)}$.
- Else $\boldsymbol{den(s) \circ X}$.

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

- 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 $\boldsymbol{f}$ 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?