CS-XXX: Graduate Programming Languages

Lecture 5 — Pseudo-Denotational Semantics

Dan Grossman
2012

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 (translater), from abstract syntax to *a different language with known semantics*

Target language is math, but we'll make it a tiny core of OCaml (hence "pseudo")

Metalanguage is math or OCaml (we'll show both)

The basic idea

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

$$string \rightarrow int$$

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

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

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

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

Now just define den in our metalanguage (math or ML), inductively over the source language abstract syntax

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 meta language or target language?
  - Translate abstract $+$ to OCaml +, (ignoring overflow)
- *When* do we denote e_1 and e_2?
  - Not a focus of the metalanguage. At "compile time".

Switching metalanguage

With OCaml as our metalanguage, ambiguities go away

But it is 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 does not have such a feature, but the point is we no longer need the abstract syntax

See `denote.ml`

Statements, w/o while

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

$den(\textbf{skip}) \quad = \quad$ `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)`

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

`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

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

```
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!

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 OCaml program using only functions, variables, ifs, constants, +, *, >, etc.

- Does not 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 $den(s)$)

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, which we will not pursue

For (1), we "lift" the *semantic domains* to include a special $\bot$

$$den(s) : (string \rightarrow int) \rightarrow ((string \rightarrow int) \cup \bot)$$

- Have to change meaning of $den(s_2) \circ den(s1)$ appropriately

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

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

Now let $den($**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 is the limit of starting with $\bot$ 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?