

CSE-505: Programming Languages

Lecture 5 — Pseudo-Denotational Semantics

Zach Tatlock
2016

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 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:

$$\mathit{string} \rightarrow \mathit{int}$$

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

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

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

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

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

Expressions

$$den(e) : (string \rightarrow int) \rightarrow int$$
$$den(c) = \text{fun } h \rightarrow c$$
$$den(x) = \text{fun } h \rightarrow h \ x$$
$$den(e_1 + e_2) = \text{fun } h \rightarrow (den(e_1) \ h) + (den(e_2) \ h)$$
$$den(e_1 * e_2) = \text{fun } h \rightarrow (den(e_1) \ h) * (den(e_2) \ h)$$

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

$$\mathit{den}(s) : (\mathit{string} \rightarrow \mathit{int}) \rightarrow (\mathit{string} \rightarrow \mathit{int})$$
$$\mathit{den}(\mathbf{skip}) \quad = \quad \mathit{fun} \ h \ -> \ h$$
$$\mathit{den}(x := e) \quad = \quad \mathit{fun} \ h \ -> \ (\mathit{fun} \ v \ -> \ \mathit{if} \ x=v \ \mathit{then} \ \mathit{den}(e) \ h \ \mathit{else} \ h \ v)$$
$$\mathit{den}(s_1; s_2) \quad = \quad \mathit{fun} \ h \ -> \ \mathit{den}(s_2) \ (\mathit{den}(s_1) \ h)$$
$$\mathit{den}(\mathbf{if} \ e \ s_1 \ s_2) \quad = \quad \mathit{fun} \ h \ -> \ \mathit{if} \ \mathit{den}(e) \ h > 0 \ \mathit{then} \ \mathit{den}(s_1) \ h \ \mathit{else} \ \mathit{den}(s_2) \ h$$

Same ambiguities; same answers

See `denote.ml`

While

$den(\mathbf{while} \ e \ s) =$	While(e,s) ->
let rec f h =	let d1=denote_exp e in
if ($den(e) \ h$)>0	let d2=denote_stmt s in
then f ($den(s) \ h$)	let rec f h =
else h in	if (d1 h)>0
f	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 $den(\mathbf{while} \ e \ s) = den(\mathbf{if} \ e \ (s; \mathbf{while} \ e \ s) \ \mathbf{skip})$ 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 $\llbracket s \rrbracket$ instead of $den(s)$)

Example: $\llbracket x := e \rrbracket \llbracket H \rrbracket = \llbracket H \rrbracket [x \mapsto \llbracket e \rrbracket \llbracket H \rrbracket]$

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 \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 **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 $X \circ \text{den}(s)$

Now let $\text{den}(\text{while } e \text{ } s)$ 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 \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?