Tail Recursion

Are these the same?

```
let range n =
  if n < 0 then
    []
  else
    n :: range (n-1)
```

```
let range n =
  let loop acc i =
    if i < 0 then
      List.rev acc
    else
      loop (i::acc) (i-1)
  in
  loop [] n
```

A

Tail call : final thing function does is a call.

Who cares ?

Well, tail calls can be made w/out allocating another stack frame... extra important when we don't have loops!

B

CSE 505 : Grad PL (Fall 2013)

Lecture 6 - Pseudo-Denotational Semantics

So far, to define what programs mean, we've been defing <u>operational semantics</u>; essentially, high-level, abstract interpreters. These interpreters are effectively just functions from AST (+ heap) to AST. The metalanguage we use to specify these semantics is a set of inference rules (on board in past lectures) or OCaml (as in homework).

What are some downsides of operational semantics?

Another style: <u>denotational semantics</u>.

In denotational semantics we reduce the meaning of something we don't know (a program) to something we do know (math, another lang, etc.).

Essentially compile (translate) from AST to another language with <u>known meaning</u> (semantics).

Normally, the target language is math. (why?) But here we'll use OCaml (hence "pseudo").

Metalanguage is math or Ocaml. We'll see both.

Basic Idea

- heaps are math/ML functions from strings to integers

$$\text{heap} \triangleq \text{string} \to \text{int}$$
$$\text{string} \to \mathbb{Z}$$

- expressions <u>denote</u> (map to) math/ML function from heaps to integers

$$\text{denote}(e) : \text{heap} \to \text{int}$$
$$(\text{sting} \to \text{int}) \to \text{int}$$

- statements denote math/ML function from heaps to heaps

$$\text{denote}(s) : \text{heap} \to \text{heap}$$
$$(\text{string} \to \text{int}) \to (\text{string} \to \text{int})$$
$$(\text{string} \to \mathbb{Z}) \to (\text{string} \to \mathbb{Z})$$

Now we need to define "den" in our metalanguage (math/ML), inductively over the source language AST.

<u>Expressions</u>

$$\mathrm{den}(e) : \overbrace{(\mathrm{string} \to \mathrm{int})}^{\mathrm{heap}} \longrightarrow \mathrm{int}$$

$$\mathrm{den}(c) = \mathrm{fun}\ h \to c$$

$$\mathrm{den}(x) = \mathrm{fun}\ h \to h\ x$$

$$\mathrm{den}(e_1 + e_2) = \mathrm{fun}\ h \to (\mathrm{den}(e_1)\ h) + (\mathrm{den}(e_2)\ h)$$

$$\mathrm{den}(e_1 * e_2) = \mathrm{fun}\ h \to (\mathrm{den}(e_1)\ h) * (\mathrm{den}(e_2)\ h)$$

Wait a second... what do these different "+" and "*" mean?

LHS "+" is just syntax.

RHS "+" is it from meta language or
target language ?

▷ abstract "+" translates to OCaml "+"

- do we need to ignore overflow?

▷ When do we denote e_1 and e_2 ?

- at "compile time"
- not a focus of the metalanguage

Q: Is GCC a semantics for C ?

Q: Is it a good semantics ?

Switch Metalanguage

With Ocaml as metalanguage, no ambiguity.

- ▷ but now hard to distinguish between "target" and "meta" languages

If denote (den(s)) is in function body, then we still have source code "around at runtime".

- ▷ After translation, should be able to "remove" defn of AST
  - totally translate everything down ~~down~~ to target language

- ▷ Can't really coerce ML to check this for us, but point is, we should not ever need AST when considering the denotation of a program.

see denote.ml

Statements (modulo while)

$$\mathrm{den}(s) : \underbrace{(\text{string} \to \text{int}) \to (\text{string} \to \text{int})}$$

heap - transformer
(operate on functions - higher order)

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

just function composition

```
den(s1 ; s2) =
    fun h -> den(s2) (den(s1) h)
```

```
den( if e s1 s2) =
  fun h ->
    if den(e) h > 0 then
        den(s1)
    else
        den(s2)
```

Similar ambiguities to expr case (e.g. <).

↳ similar answers

$\boxed{\text{see denote.ml}}$

While

(WRONG)

den(while e s) =

den(if e (s; while e s) skip)

Why?

- circular! will never "bottom out" (can't be ifs all the way down... ∞ - unrolling)

What should we do instead?

- use recursive func

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

Q: Is this any better?

Well, at least translation terminates!

In OCaml, be careful not to leave any dangling AST remnants...

```
| 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
```

get all translation done @ "compile time"

Avoiding Pitfalls

- A denotational semantics should "eagerly" translate entire program
  - e.g. both branches of if

- A denotational semantics should "terminate" (translation always successful)
  - avoid circularity in translations
  - the result (target, output, etc.) can use recursion
  - should never produce ∞ code...
    - (like in our WRONG version of while)

Tying Up Loose Ends

```
let denote_prog s =
    let d = denote_stmt s in
    fun () -> (d (fun x -> 0)) "ans"
```

▷ Compilation

```
let asm = denote_prog (parse source)
```

▷ Execution

```
print_int (asm ())
```

"asm" completely in target language:

- OCaml program using only functions, variables, ifs, constants, +, *, <
- does <u>not</u> use anything from AST

Sketch non-pseudo denotational semantics

In "real" versions, target language is math.

Use $[\![s]\!]$ notation for den(s).

Example

$$[\![x := e]\!] [\![H]\!] = [\![H]\!] [x \mapsto [\![e]\!] [\![H]\!]]$$

A couple major challenges arise when we go to handle while:

(1) Normal mathematical functions do not diverge; unclear how to translate "while 1 skip"

(2) The denotation of loops cannot be circular.

Handling divergence:

- "lift" our semantic domains to include a special non-termination value "$\perp$" pronounced "bottom"
  - used just in codomain (range)
- need to update composition

  $den(s_2) \circ den(s_1)$

Avoiding circularity:

- SO MUCH WORK
- Define a meta function F from lifted heap transformers to lifted heap transformers

This is an under-approximation of the while's behavior. The more we iterate, the better the approx:

handles up to 2 iters

$$\triangleright\ W_2 = \mathcal{F}(W_1)$$

$$= \text{fun } h \to$$

$$\quad \text{if den}(e)\ h > 0 \text{ then}$$

$$\quad\quad W_1 \circ (\text{den}(s)\ h)$$

$$\quad \text{else}$$

$$\quad\quad h$$

Now we want to consider behavior of $\mathcal{F}$ in the limit (as # of nestings $\to \infty$)

$\mathcal{F}(d) = $ fun $h \rightarrow$

if den(e) $h > 0$ then

$d \circ (\text{den}(s)\ h)$

else

h

Now den(while e s) = least fixed point of $\mathcal{F}$

- start w/ $W_0 = $ fun $h \rightarrow \bot$
- $W_n = \mathcal{F}(W_{n-1})$

– $W_1 = \mathcal{F}(W_0)$

= fun $h \rightarrow$

if den(e) $h > 0$ then

$W_0 \circ (\text{den}(s)\ h)$ // becomes $\bot$

else

h

∴ Turns out there is a limit. (+ fixpoint then)

it's

▷ lots of work

Look Up:

The Fixed-Point Theorem

CPO (complete partial order)

monotonic functions

Whew! Summary:

▷ seen syntax, op + denot semantics

▷ connections to interp / compilers

▷ Coming up: equivalence

▷ Q: which is better? op or denote sem?

Denotatial Semantics

Equivalence:

For p_1 and p_2, $[\![p_1]\!] = [\![p_2]\!]$ iff p_1 and p_2 are observationally equivalent.

Compositionality:

The denotation of a program should be constructed from the denotations of its parts.

$$den(e_1 + e_2) = den(+)(den(e_1), den(e_2))$$

$$[\![e_1 + e_2]\!][\![H]\!] = [\![+]\!]([\![e_1]\!][\![H]\!], [\![e_2]\!][\![H]\!])$$

What if we have functions in exprs?

- how does it affect op sem?
- denot. sem?

What if func args evaled in parallel?

A

Dana Scott:

"Semantics need not determine the implementation, but they should provide for showing an implementation is correct."

B