Tail Recursion

Are these the same?

\[
\begin{align*}
\text{let } \text{range } n &= \\
\text{if } n < 0 \text{ then } &[] \\
\text{else } &n :: \text{range } (n-1)
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{range } n &= \\
\text{let } \text{loop } \text{acc } i &= \\
\text{if } i < 0 \text{ then } &\text{List.rev } \text{acc} \\
\text{else } &\text{loop } (i::\text{acc}) (i-1) \\
\text{in } &\text{loop } [] n
\end{align*}
\]
don't have loops!

frame... extra important when we
without allocating another stack.
well, tail calls can be made
Who cares?

Is a call:

Tail call: final thing function does
CSE 505: Grad PL (Fall 2013)

Lecture 6 - Pseudo-Denotational Semantics

So far, to define what programs mean, we've been defining operational semantics; 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: **denotational semantics**.

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 **known meaning** (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.

\[
\text{heap} : \text{string} \rightarrow \text{int}
\]
\[
\text{string} \rightarrow \mathbb{Z}
\]

- Expressions denote (map to) math/ML function from heaps to integers.

\[
\text{denote(e)} : \text{heap} \rightarrow \text{int}
\]
\[
(\text{string} \rightarrow \text{int}) \rightarrow \text{int}
\]

- Statements denote math/ML function from heaps to heaps.

\[
\text{denote(s)} : \text{heap} \rightarrow \text{heap}
\]
\[
(\text{string} \rightarrow \text{int}) \rightarrow (\text{string} \rightarrow \text{int})
\]
\[
(\text{string} \rightarrow \mathbb{Z}) \rightarrow (\text{string} \rightarrow \mathbb{Z})
\]
Now we need to define "den" in our metalanguage (math/ML), inductively over the source language AST.

Expressions

\[
\text{heap}
\]

\[
\text{den}(e) : (\text{string} \Rightarrow \text{int}) \Rightarrow \text{int}
\]

\[
\text{den}(c) = \text{fun } h \Rightarrow c
\]

\[
\text{den}(x) = \text{fun } h \Rightarrow h \cdot x
\]

\[
\text{den}(e_1 + e_2) = \text{fun } h \Rightarrow (\text{den}(e_1) \cdot h) + (\text{den}(e_2) \cdot h)
\]

\[
\text{den}(e_1 \cdot e_2) = \text{fun } h \Rightarrow (\text{den}(e_1) \cdot h) \cdot (\text{den}(e_2) \cdot h)
\]

Wait a second... what do these different "+" and "\cdot" 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 e₁ and e₂?
  - 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 (denCI) 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.
Statements (modulo while)

\[ \text{den}(s) : (\text{string} \rightarrow \text{int}) \rightarrow (\text{string} \rightarrow \text{int}) \]

heap transformer
(operate on functions - higher order)

\[
\text{den}(s) = \text{fun } h \rightarrow h
\]

\[
\text{den}(x := e) = \text{fun } h \rightarrow \text{fun } v \rightarrow \text{if } v = x \text{ then } \text{den}(e) \ h \text{ else } h \ v
\]

\[
\text{just function composition}
\]

\[
\text{den}(s_1 ; s_2) = \text{fun } h \rightarrow \text{den}(s_2) \ (\text{den}(s_1) \ h)
\]
\[
\text{den}(\text{if } e \text{ s}_1 \text{ s}_2) =
\begin{align*}
\text{fun } h & \rightarrow \\
\text{if } \text{den}(e) h > 0 & \text{ then } \\
\text{den}(s_1) & \\
\text{else } \\
\text{den}(s_2) &
\end{align*}
\]

Similar ambiguities to expr case (e.g. <). 
\Rightarrow \text{ similar answers}

see denote.ml
While

\[ \text{den}(\text{while } e \neq s) = \]
\[ \text{den}(\text{if } e \neq s; \text{ while } e \neq s \text{ skip}) \]

Why?
- circular! will never "bottom out"
  (can't be if's all the way down...
  \(\infty\) - unrolling)

What should we do instead?
- use recursive func
\[
den(\text{while } e \text{ s}) = \\
\text{let rec } f \ h = \\
\quad \text{if } \ den(e) \ h \ > \ 0 \ \text{then} \\
\quad \quad f \ (\ den(s) \ h) \\
\quad \text{else} \\
\quad \ h \\
\quad \text{in} \\
\quad f \\
\]

Q: Is this any better?

Well, at least it terminates!
In OCaml, be careful not to leave any dangling AST remnants...

```ocaml
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
```

[11]
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 d = denote-prog s in
\( \text{fun}\ (x : \text{unit})\ p \rightarrow (\text{filter} (\lambda a. a \neq x) p) \text{ in} \)

let asm = denote-prog (parse source) in

\(\triangle\) Compilation

\(\triangle\) Execution

\(\triangle\) Print-int (asm c)
"asm" completely in target language:
- OCaml program using only functions, variables, ifs, constants, +, *, <
- does not use anything from AST

Sketch non-pseudo denotational semantics

In "real" versions, target language is math.

Use \([s]\) notation for \(\text{den}(s)\).
Example

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

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

1. Normal mathematical functions do not diverge; unclear how to translate "while I skip"

2. The denotation of loops cannot be circular.
Handling divergence:

- "lift" our semantic domains to include a special non-termination value "⊥
  pronounced "bottom" - used just in codomain (range)
- need to update composition
  \[ \text{den}(s_2) \circ \text{den}(s_1) \]

Avoiding circularity:

- so much work
- Define a meta function \( F \) from \( v \) heap transformers to lifted heap transformers
This is an under-approximation of the while's behavior. The more we iterate, the better the approx:

\[ D W_2 = F (w, h) \]

\[ = \text{fun~} h \Rightarrow \]

\[ \text{if } \text{den(e)} \cdot h > 0 \text{ then} \]

\[ W_1 \circ (\text{den(s)} \cdot h) \]

\[ \text{else} \]

\[ h \]

Now we want to consider behavior of \( F \) in the limit (as # of nestings \( \to \infty \))
\( F(d) = \text{fun } h \Rightarrow \)

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

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

else

\( h \)

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

\( \triangleright \) start w/ \( W_0 = \text{fun } h \Rightarrow \bot \)

\( \triangleright \) \( W_n = F(W_{n-1}) \)

\( - W_1 = F(W_0) \)

\( = \text{fun } h \Rightarrow \)

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

\( W_0 \circ (\text{den}(s) \cdot h) \triangleright \bot \)

else

\( h \)
Turns out there is a limit. (fixpoint theory)

> 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 denot sem?
What is fun arg equality in parallel?

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

What if we have functions in exprs?

\[
\langle e_1 + e_2 \mid H \rangle = \langle e_1 \mid H \rangle + \langle e_2 \mid H \rangle
\]

\[
\text{den} (e_1 + e_2) = \text{den} (\text{+} (\text{den} (e_1), \text{den} (e_2))
\]

Parts.

Constructs detd from the denotations of its.

The denotation of a program should be

Compositionality:

If \( p_1 \) and \( p_2 \) are observationally equivalent,

\[
[p_1] = [p_2]
\]

For \( p_1 \) and \( p_2 \),

Equivalence:

Denotational Semantics