LECTURE 04

Theory and Design of PL (CS 538)

February 3, 2020

LAMBDA CALCULUS BASICS

SOME TERMINOLOGY

- You may see several different names:
  - Programs
  - Expressions
  - Terms
- For lambda calculus, these all mean the same thing

"RUNNING" A LAMBDA EXPRESSION

- Given a lambda calculus program, how to run it?
    1. Figure out where parentheses go
    2. Substitute fn argument into fn body
    3. Repeat until we reach a value

1. FIGURE OUT WHERE PARENTHESES GO

- Function application is *left-associative*
- Example: $e_1\ e_2\ e_3$ means $(e_1\ e_2)\ e_3$
  - Read: call e_1 with e_2, then with e_3
- Not the same: $e_1\ (e_2\ e_3)$
  - Read: call e_2 with e_3, then call e_1

2. SUBSTITUTE ARGUMENT INTO BODY

- Example: $(\lambda x.e)\ v$ where v is a value
- Replace all* x's in e with v, remove λx.
  - Read: call function with argument v
- Example: $(\lambda x.x + 1)\ 5$
  - Replace x with 5, remove λx.
  - Result: $5 + 1$, steps to 6

3. KEEP SUBSTITUTING UNTIL DONE

1. Order: outside-to-inside
2. Operate on left-most term until it is $\lambda x.e$
3. Turn to argument (right-most term)
   - If eager evaluation, operate on argument
   - If lazy evaluation, substitute argument into e
4. Never substitute "under" lambdas
   - Don't substitute for y: $\lambda x.((\lambda y.e_1)\ x)$

LET'S DO AN EXAMPLE

- Start: $((\lambda a.a)\ \lambda b.\lambda c.\lambda d.d\ b\ c)\ 1\ 2\ \lambda x.\lambda y.x + y$
- $= ((((\lambda a.a)\ \lambda b.\lambda c.\lambda d.(d\ b)\ c)\ 1)\ 2)\ (\lambda x.\lambda y.x + y)$
  - $\to (((\lambda b.\lambda c.\lambda d.(d\ b)\ c)\ 1)\ 2)\ (\lambda x.\lambda y.x + y)$
  - $\to ((\lambda c.\lambda d.(d\ 1)\ c)\ 2)\ (\lambda x.\lambda y.x + y)$
  - $\to (\lambda d.(d\ 1)\ 2)\ (\lambda x.\lambda y.x + y)$
  - $\to ((\lambda x.\lambda y.x + y)\ 1)\ 2$
  - $\to (\lambda y.1 + y)\ 2$
  - $\to 1 + 2 \to 3$

FREE VERSUS BOUND VARIABLES

- Free variable: introduced by outer λ
- Bound variable: not introduced by outer λ
- Example: $z\ \lambda x.z + x$
  - x is a bound variable (under λx)
  - z is a free variable (not under λz)
- When substituting, only replace bound variable
- Example: $(\lambda x.(\lambda x.x + 1))\ 5$ steps to $\lambda x.x + 1$
  - Inner x bound by the inner λx, not the outer one

SPECIFYING PROGRAM BEHAVIORS

HELP COMPILER WRITERS

- For real languages: multiple implementations
  - C/C++: gcc, clang, icc, compcert, vc++, …
  - Python: CPython, Jython, PyPy, …
  - Ruby: YARV, JRuby, TruffleRuby, Rubinius, …
- Should agree on what programs are supposed to do!

DESIGN OPTIMIZATIONS

- Compilers use optimizations to speed up code
  - Loops: fission and fusion, unrolling, unswitching
  - Common subexpression, dead code elimination
  - Inlining and hoisting
  - Strength reduction
  - Vectorization
  - ...
- Optimizations shouldn’t affect program behavior!

PROVE PROGRAMS SATISFY CERTAIN PROPERTIES

- Before we can prove anything about programs, we first need to formalize what programs do
- Example: equivalence
  - Which programs are equivalent?
  - Which programs aren't equivalent?

HOW TO SPECIFY BEHAVIORS?

PROGRAM SEMANTICS

- Ideal goal: *describe programs mathematically*
  - Aiming for a fully precise definition
- But: no mathematical model is perfect
  - Programs run on physical machines in real life
- Challenge: which aspects should we model?

MANY APPROACHES

- Denotational semantics
  - Translate programs to mathematical functions
- Axiomatic semantics
  - Analyze pre-/post-conditions of programs
- **Operational semantics**
  - Model how programs step

> *Principle: program behavior should be defined by behavior of its components*

OPERATIONAL SEMANTICS

PROGRAMS MAKE STEPS

- Model how a program is evaluated
- Benefits:
  - Closer correspondence with implementation
  - General: most programs “step”, in some sense
- Drawbacks:
  - A lot of details, models all the steps
  - Overkill if we just care about input/output

VALUES AND EXPRESSIONS

- Programs may or may not be able to step
- Can step: *redexes* (reducible expresisons)
- Can't step:
  - *Values*: valid results
  - *Stuck terms*: invalid results ("runtime errors")

IN LAMBDA CALCULUS

- Values: these things *do not* step, they are done

```
val = 𝔹 | ℤ | var | λ var . expr
```

- Expressions: these things *may* step

```
expr = 𝔹 | ℤ
     | λ var . expr | expr expr
     | add(expr, expr) | sub(expr, expr)
     | and(expr, expr) | or(expr, expr)
     | if expr then expr else expr | ...
```

- Stuck terms: not values, but *can't* step (error)
  - `true 1`
  - `1 + false`

HOW TO DEFINE OPERATIONAL SEMANTICS?

WANT TO DEFINE NEW RELATIONS

- $\mathrm{R}(\mathrm{e}, \mathrm{v})$: "Program e steps to value v"
- $\mathrm{S}(\mathrm{e}, \mathrm{e}')$: "Program e steps to program e'"
- As PL designer: we get to *define* R and S
  - But what does a definition look like?

INFERENCE RULES

- Basic idea: we write down a set of inference rules
- Components of a rule
  - Above the line: zero-or-more assumptions
  - Below the line: one conclusion
- Meaning of a rule
  - If top thing(s) hold, then bottom thing holds
  - If no top things: bottom thing holds

EXAMPLE: ISDOUBLE

BIG-STEP SEMANTICS

IDEA: DESCRIBE PROGRAM RESULT

- Useful for language specifications
- Don't describe intermediate steps
- Write $\mathrm{e} \Downarrow \mathrm{v}$ if program e evaluates to value v

Language designer defines when $\mathrm{e} \Downarrow \mathrm{v}$

EXAMPLE

HOW TO APPLY FUNCTIONS?

- Eager evaluation
  - If $e_1 \Downarrow \lambda x.e_1'$, and
  - If $e_2 \Downarrow v$, and
  - If $e_1'[x \mapsto v] \Downarrow v'$,
  - Then: $e_1\ e_2 \Downarrow v'$

HOW TO APPLY FUNCTIONS?

- Lazy evaluation
  - If $e_1 \Downarrow \lambda \mathbf{x}.e_1'$, and
  - If $e_1'[\mathbf{x} \mapsto e_2] \Downarrow \mathbf{v}$,
  - Then: $e_1\ e_2 \Downarrow \mathbf{v}$

IN HASKELL?

- Recall tuple and non-terminating functions:

```
fst (x, y) = x
snd (x, y) = y

loopForever x = loopForever x -- never terminates
```

- What if we try to project from a bad tuple?

```
badFst = fst (loopForever 42, 0) + 1 -- Never returns
badSnd = snd (loopForever 42, 0) + 1 -- Returns 1!
```

EAGER EVALUATION

- When passing arguments to function, first evaluate argument all the way
- Also known as *call-by-value (CBV)*
- If argument doesn't terminate, then function call doesn't terminate

```
badFst = fst (loopForever 42, 0) + 1 -- Never returns under CBV
badSnd = snd (loopForever 42, 0) + 1 -- Never returns under CBV
```

LAZY EVALUATION

- Only evaluate arguments *when they are needed*
- Also known as *call-by-name* (CBN)
- This is Haskell's evaluation order

```
badFst = fst (loopForever 42, 0) + 1 -- Never returns under CBN
badSnd = snd (loopForever 42, 0) + 1 -- Returns 0 under CBN
```

FUN WITH LAZINESS

- Can write various kinds of infinite data
- Values are computed *lazily*: only when needed

```haskell
lotsOfOnes :: [Int]
lotsOfOnes = 1 : lotsOfOnes  -- [1, 1, ...

firstOne   = head lotsOfOnes -- Returns 1

onesAndTwos :: [Int]         -- [1, 2, 1, 2, ...
onesAndTwos = x where x = 1 : y
                      y = 2 : x

firstTwo = head $ tail onesAndTwos -- Returns 2

fibonacci :: [Int]           -- [1, 1, 2, 3, ...
fibonacci = 1 : 1 : zipWith (+) fibonacci (tail fibonacci)
```

SMALL-STEP SEMANTICS

IDEA: DESCRIBE PROGRAM STEPS

- More fine-grained, helpful for implementation
- If e steps to e' in one step, write: $e \to e'$
- If e steps to e' in zero or more steps: $e \to^* e'$

EXAMPLE

RECURSION

FIXED POINT OPERATION

- Idea: special expression for recursive definitions
- Should allow definition to “make recursive call”
- *Fixed point expression*: defined in terms of itself

```
expr = ... | fix var . expr
```

HOW DOES THIS EVALUATE?

- In $\mathrm{fix}\ \mathrm{f}.\ \mathrm{e}$:
  - The variable f represents recursive call
  - The body e can make recursive calls via f
- Small-step:

$$\mathrm{fix}\ \mathrm{f}.\ \mathrm{e} \to \mathrm{e}[\mathrm{f} \mapsto \mathrm{fix}\ \mathrm{f}.\ \mathrm{e}]$$

- Big-step:
  - If $\mathrm{e}[\mathrm{f} \mapsto \mathrm{fix}\ \mathrm{f}.\ \mathrm{e}] \Downarrow \mathrm{v}$,
  - Then: $\mathrm{fix}\ \mathrm{f}.\ \mathrm{e} \Downarrow \mathrm{v}$

HOW TO USE THIS THING?

- Suppose: want to model factorial function:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

- We can model as the following expression:

$$\text{factorial} = \text{fix f. } \lambda \text{n. if n} = 0 \text{ then } 1 \text{ else n} * (\text{f } (\text{n} -$$

TESTING IT OUT

- Evaluating factorial 5:
  - $\rightarrow [\lambda n.\ \text{if } n = 0 \text{ then } 1 \text{ else } n * ((\text{fix } f \ldots)\ (n - 1))$
  - $\rightarrow \text{if } 5 = 0 \text{ then } 1 \text{ else } 5 * ((\text{fix } f \ldots)\ (5 - 1))$
  - $\rightarrow^* 5 * ((\text{fix } f \ldots)\ 4)$
  - $\ldots$
  - $\rightarrow^* 5 * 4 * 3 * 2 * (\text{if } 0 = 0 \text{ then } 1 \text{ else } \ldots)$
  - $\rightarrow 5 * 4 * 3 * 2 * 1 \rightarrow^* 120$