CSC 520, Spring 2020

Principles of Programming Languages

Michelle Strout

THE UNIVERSITY OF ARIZONA

Version: 1

April 22, 2020

Plan

- **Announcements**
  - HW9 is due today
  - HW10 was posted last Friday and is due Wednesday April 29th
- **Last time**
  - Six questions about uSmalltalk
  - Message passing
  - Dynamic dispatch
  - Predefined number classes
- **Today**
  - Lambda Calculus Overview
  - Programming in the Lambda Calculus
  - Operational Semantics of Lambda Calculus

What is a calculus? Manipulation of syntax

THE UNIVERSITY OF ARIZONA
Computer Science

- **Demonstration of differential calculus: reduce**

```
d/dx (x + y)^2
```

- **Rules**

```
d/dx k = 0
d/dx x = 1
d/dx y = 0 where y != x

d/dx (u + v)
 = d/dx u + d/dx v

d/dx (u * v)
 = u* d/dx v + v* d/dx u

d/dx (e^n)
 = n * e^(n-1) * d/dx e
```

Eval is reduction to normal form

```
d/dx (x + y)^2

2 · (x + y) · d/dx (x + y)

2 · (x + y) · (d/dx x + d/dx y)

2 · (x + y) · (1 + d/dx y)

2 · (x + y) · (1 + 0)  2 · (x + y) · 1

2 · (x + y)
```

Why study lambda calculus?

THE UNIVERSITY OF ARIZONA
Computer Science

- Theoretical underpinnings for most programming languages (all of those studied this semester)
- Church-Turing Thesis: Any computatable operator can be expressed as an encoding in the lambda calculus
- Test bench for new language features

Simplest Reasonable PL

- **Just application, abstraction, and variables**

- **Only three syntactic forms:**

```
M ::= x | \x.M | M M′
```

- **Everything is programming with functions**
  - Everything is curried (only one parameter per function abstraction)
  - Application associates to the left
  - Arguments are not evaluated

```
(\x.\y.x) M N → (\y.M) N → M
```

- **Crucial: argument N is never evaluated (infinite loop is possible)**

Programming in Lambda Calculus

THE UNIVERSITY OF ARIZONA
Computer Science

- **Everything is continuation-passing style**
  - Q: Who remembers the boolean equation solver?
  - Q: What classes of results could it produce?
  - Q: How were the results delivered?
  - Q: How did we do Boolean's in smalltalk?

Coding Booleans

THE UNIVERSITY OF ARIZONA
Computer Science

- **Booleans take two continuations**

```
true  = \t.\f.t
false = \t.\f.f

// laws for if
if true then N else P = N
if false then N else P = P

if = \b.\t.\e.b t e
```

- **Your turn: implement not**
  - Algebraic laws for not: what are the forms of the input data?

```
not  true  = false
not  false = true
```

  - Code for not

```
not  = \b.b false true
```

Coding Pairs/Cons

- **Questions to consider**
  - How many ways can pairs be created? pair(x,y)
  - How many continuations? One continuation for pair data ctor
  - What information does pair expect? Expects two pieces of info

- **Implementing pair, fst, and snd**
  - Algebraic laws

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

  - Code

```
pair = \x.\y.\f.f x y
fst = \p.p (\x.\y.x)
snd = \p.p (\x.\y.y)
```

Pairs

- Just like in Scheme, we can define pairs — these allow us to construct data structures such as lists and trees.
- The definition of Pair below is similar to a dotted pair notation (or `cons`) in Scheme.
- Head and Tail correspond to `car` and `cdr`, Nil is a special constant.

$$\text{Pair} \equiv (\lambda a.(\lambda b.(\lambda f.((f \;\; a) \;\; b))))$$

$$\text{Head} \equiv (\lambda g.(g \;\; (\lambda a.(\lambda b.a))))$$

$$\text{Tail} \equiv (\lambda g.(g \;\; (\lambda a.(\lambda b.b))))$$

$$\text{Nil} \equiv (\lambda x.(\lambda a.(\lambda b.a)))$$

Slide content credits: Christian Collberg

Pairs...

- We can construct a pair (p, q) (or $(p.q)$ in Scheme notation) like this:

$$\text{Pair} \equiv (\lambda a.(\lambda b.(\lambda f.((f \;\; a) \;\; b))))$$

$$((\text{Pair} \;\; p) \;\; q) =$$

$$(((\lambda a.(\lambda b.(\lambda f.((f \;\; a) \;\; b)))) \;\; p) \;\; q) \Rightarrow_\beta$$

$$((\lambda b.(\lambda f.((f \;\; p) \;\; b))) \;\; q) \Rightarrow_\beta$$

$$(\lambda f.((f \;\; p) \;\; q))$$

Pairs...

THE UNIVERSITY OF ARIZONA. Computer Science

- We can verify that Head works as specified:

$$\text{Pair} \equiv (\lambda a.(\lambda b.(\lambda f.((f \;\; a) \;\; b))))$$
$$\text{Head} \equiv (\lambda g.(g \;\; (\lambda a.(\lambda b.a))))$$

$$((\text{Pair} \;\; p) \;\; q) = (((\lambda a.(\lambda b.(\lambda f.((f \;\; a) \;\; b)))) \;\; p) \;\; q) \Rightarrow_\beta$$

$$((\lambda b.(\lambda f.((f \;\; p) \;\; b))) \;\; q) \Rightarrow_\beta (\lambda f.((f \;\; p) \;\; q))$$

$$(\text{Head} \;\; ((\text{Pair} \;\; p) \;\; q)) = (\text{Head} \;\; (\lambda f.((f \;\; p) \;\; q))) =$$

$$((\lambda g.(g \;\; (\lambda a.(\lambda b.a)))) \;\; (\lambda f.((f \;\; p) \;\; q))) \Rightarrow_\beta$$

$$((\lambda f.((f \;\; p) \;\; q)) \;\; (\lambda a.(\lambda b.a))) \Rightarrow_\beta (((\lambda a.(\lambda b.a)) \;\; p) \;\; q) \Rightarrow_\beta^* p$$

Coding Lists

- **Questions to consider**
  - How many ways can pairs be created? (empty list or cons)
  - How many continuations?
  - What information does pair expect?

```
cons y ys n c = c y ys
nil       n c = n

car   xs = xs error (\y.\ys.y)
cdr   xs = xs error (\y.\ys.ys)
null? xs = xs true (\y.\ys.false)

cons = \y.\ys.\n.\c.c y ys
nil = \n.\c.n

car   = \xs.xs error (\y.\ys.y)
cdr   = \xs.xs error (\y.\ys.ys)
null? = \xs.xs true  (\y.\ys.false)
```

null? applied to nil

```
cons = \y.\ys.\n.\c.c y ys
nil = \n.\c.n

car   = \xs.xs error (\y.\ys.y)
cdr   = \xs.xs error (\y.\ys.ys)
null? = \xs.xs true  (\y.\ys.false)

null? nil → (\xs.xs   true (\y.\ys.false)) nil
          →      nil  true (\y.\ys.false)
          → (\n.\c.n) true (\y.\ys.false)
....
          → true
```

Coding numbers

- **Wikipedia good: “Church numerals”**
- **Key idea: the value of a number is the number of times it applies its argument function**

Church Numerals

THE UNIVERSITY OF ARIZONA
Computer Science

Encoding natural numbers as lambda-terms

$$\begin{aligned}
\text{zero} &= \lambda f.\lambda x.x \\
\text{one} &= \lambda f.\lambda x.f\ x \\
\text{two} &= \lambda f.\lambda x.f\,(f\ x) \\
\text{n} &= \lambda f.\lambda x.f^{(n)}x \\
\text{succ} &= \lambda n.\lambda f.\lambda x.f\,(n\ f\ x) \\
\text{plus} &= \lambda n.\lambda m.n\ \text{succ}\ m \\
\text{times} &= \lambda n.\lambda m.n\ (\text{plus}\ m)\ \text{zero}
\end{aligned}$$

Church's Numerals — $\mathsf{succ}\ldots$

$$\mathsf{succ} \equiv (\lambda n.(\lambda f.(\lambda x.(f \;\; ((n \;\; f) \;\; x)))))$$
$$2 \equiv (\lambda g.(\lambda y.(g \;\; (g \;\; y))))$$
$$3 \equiv (\lambda f.(\lambda x.(f \;\; (f \;\; (f \;\; x)))))$$

$(\mathsf{succ} \;\; 2) \Rightarrow$

$$((\lambda n.(\lambda f.(\lambda x.(f \;\; ((n \;\; f) \;\; x))))) \;\; (\lambda g.(\lambda y.(g \;\; (g \;\; y))))) \Rightarrow_\beta$$

$$(\lambda f.(\lambda x.(f \;\; (((\lambda g.(\lambda y.(g \;\; (g \;\; y)))) \;\; f) \;\; x)))) \Rightarrow_\beta$$

$$(\lambda f.(\lambda x.(f \;\; ((\lambda y.(f \;\; (f \;\; y))) \;\; x)))) \Rightarrow_\beta$$

$$(\lambda f.(\lambda x.(f \;\; (f \;\; (f \;\; x))))) \equiv 3$$

-Spring 2005—23 [5]

Church Numerals in lambda calculus

```
zero = \f.\x.x;
succ = \n.\f.\x.f (n f x);
plus = \n.\m.n succ m;
times = \n.\m.n (plus m) zero;
...

-> four;
\f.\x.f (f (f (f x)))

-> three;
\f.\x.f (f (f x))

-> times four three;
\f.\x.f (f (f (f (f (f (f (f (f (f (f (f x)))))))))))
```

Taking Stock

- **So far**
  - bools
  - pairs
  - lists
  - numbers

- **What is missing?**
  - Recursive functions
  - We don't need letrec or val-rec
  - Instead, use the Y-combinator = `\f.(\x.f (x x))(\x.f (x x))`

Operational semantics of lambda calculus

THE UNIVERSITY OF ARIZONA
Computer Science

- New kind of semantics: small step
- New judgement form

```
M → N ("M reduces to N in one step")
```

- No context!!
- No turnstile!! |-
- Just reducing terms == calculus

Reduction rules

Central rule based on substitution

$$\frac{}{(\lambda x.M)N \xrightarrow{\beta} M[x \mapsto N]} \quad \text{(BETA)}$$

Structural rules: Beta-reduce anywhere, any time

$$\frac{N \xrightarrow{\beta} N'}{MN \xrightarrow{\beta} MN'} \qquad \frac{M \xrightarrow{\beta} M'}{MN \xrightarrow{\beta} M'N} \qquad \frac{M \xrightarrow{\beta} M'}{\lambda x.M \xrightarrow{\beta} \lambda x.M'}$$

Bound versus Free variables

```
Same? Yes, both bound
\x.\y.x
\w.\z.w

Same? No, z is free in first
one and bound in second
\x.\y.z
\w.\z.z
```

Free variables

$$\frac{}{x \text{ is free in } x}$$

$$\frac{x \text{ is free in } M \vee x \text{ is free in } N}{x \text{ is free in } MN}$$

$$\frac{x \text{ is free in } M \qquad x \neq x'}{x \text{ is free in } \lambda x'.M}$$

Capture-avoiding substitution

$$
\begin{aligned}
x[x \mapsto M] &= M \\
y[x \mapsto M] &= y \\
(YZ)[x \mapsto M] &= (Y[x \mapsto M])(Z[x \mapsto M]) \\
(\lambda x.Y)[x \mapsto M] &= \lambda x.Y \\
(\lambda y.Z)[x \mapsto M] &= \lambda y.Z[x \mapsto M] \\
& \quad \text{if } x \text{ not free in } Z \text{ or } y \text{ not free in } M \\
(\lambda y.Z)[x \mapsto M] &= \lambda w.(Z[y \mapsto w])[x \mapsto M] \\
& \quad \text{where } w \text{ not free in } Z \text{ or } M
\end{aligned}
$$

Last transformation is renaming of bound variables

Renaming of bound variables

So important it has its own Greek letter:

$$\frac{w \text{ not free in } Z}{\lambda y.Z \xrightarrow{\alpha} \lambda w.(Z[y \mapsto w])} \quad (\textsc{Alpha})$$

Also has structural rules

Summary

- **Lambda calculus is Turing Complete**
- **Essence of most programming languages**
- **Evaluation proceeds by substituting arguments for formal variables (beta reduction)**
  - **Definition of free variables**
  - **Alpha-conversion allows bound variables to be renamed**