Lambda Calculus and Lazy Evaluation (based on material due to Graham Hutton)

1

- PS5?
- Project proposal?
- Plan: shorter lecture and “labs” today and on Friday so you can work and ask questions on monoids, foldables, monads, PS5 and Project proposal
- Next week: type inference

2

Outline

- Pure lambda calculus, a review
  - Syntax and semantics
  - Free and bound variables
  - Rules (alpha rule, beta rule)
  - Normal forms
  - Reduction strategies
- Lazy evaluation in Haskell

3

Syntax of Pure Lambda Calculus

λ-calculus formulae (e.g., $\lambda x.\ x\ y$) are called expressions or terms

$$E ::= x \mid (\lambda x.\ E_1) \mid (E_1\ E_2)$$

A λ-expression is one of

- Variable: x
- Abstraction (i.e., function definition): $\lambda x.\ E_1$
- Application: $E_1\ E_2$

4

Syntactic Conventions

Parentheses may be dropped from "stand-alone" terms **(E_1 E_2)** and **(λx. E)**

E.g., **(f x)** may be written as **f x**

Function application groups from left-to-right (i.e., it is left-associative)

E.g., **x y z** abbreviates **((x y) z)**

E.g., **E_1 E_2 E_3 E_4** abbreviates **(((E_1 E_2) E_3) E_4)**

Parentheses in **x (y z)** are necessary! Why?

Application has higher precedence than abstraction

Another way to say this is that the scope of the dot extends as far to the right as possible

E.g., **λx. x z** = **λx. (x z)** = **(λx. (x z))** = **(λx. (x z))** ≠ **((λx. x) z)**

WARNING: This is the most common syntactic convention (e.g., Pierce 2002). However, some books give abstraction higher precedence; you might have seen that different convention

Semantics of Lambda Calculus

An expression has as its meaning the value that results after evaluation is carried out

Free and Bound Variables

Abstraction **(λx. E)** is also referred as binding

Variable **x** is said to be bound in **λx. E**

The set of free variables of **E** is the set of variables that appear unbound in **E**

Defined by cases on **E**

- Var **x**: free(**x**) = {**x**}
- App **E_1 E_2**: free(**E_1 E_2**) = free(**E_1**) U free(**E_2**)
- Abs **λx. E**: free(**λx.E**) = free(**E**) - {**x**}

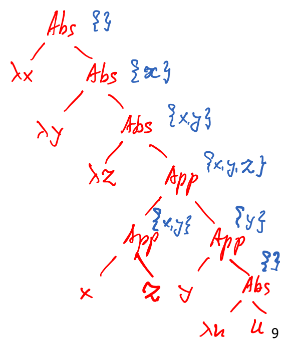
A variable **x** is bound if it is in the scope of a lambda abstraction: as in **λx. E**

Variable is free otherwise

1. (λx. x) y

2. (λz. z z) (λx. x)

3. λx.λy.λz. x z (y (λu. u))



We must take free and bound variables into account when reducing expressions

E.g., **(λx.λy. x y) (y w)**

First, rename bound **y** in **λy. x y** to **z**: **λz. x z**

(λx.λy. x y) (y w) → (λx.λz. x z) (y w)

Second, apply the reduction rule that substitutes **(y w)** for **x** in the body **(λz. x z)**

(λz. x z) [(y w)/x] → (λz. (y w) z) = λz. y w z

Substitution, formally

- **(λx.E) M → E[M/x]** replaces all free occurrences of **x** in **E** by **M**
- **E[M/x]** is defined by cases on **E**:
  - Var: **y[M/x]** = **M** if **x = y**
    y[M/x] = **y** otherwise
  - App: **(E_1 E_2)[M/x]** = **(E_1[M/x] E_2[M/x])**
  - Abs: **(λy.E_1)[M/x]** = **(λy.E_1)** if **x = y**
    (λy.E_1)[M/x] = **λz.((E_1[z/y])[M/x])** otherwise, where **z** NOT in free(**E_1**) U free(**M**) U {**x**}

(λx.λy. x y) (y w)

→ **(λy. x y)[(y w)/x]**

→ **λ1_. (((x y)[1_/y])[(y w)/x])**

→ **λ1_. ((x 1_)[(y w)/x])**

→ **λ1_. ((y w) 1_)**

→ **λ1_. y w 1_**

Exercise

```
data Lexp = Var String
          | App Lexp Lexp
          | Lam String Lexp
```

- Write a Haskell function `freshVars` that takes a list of expressions and returns an infinite list of potential fresh variables, excluding variables that occur free in some expressions

```
import Data.List

freeVars :: Lexp -> [String]
freeVars …

freshVars :: [Lexp] -> [String]
freshVars exprs = map show [1..] \\ (exprs >>= freeVars)
```


Rules (Axioms) of Lambda Calculus

α rule (α-conversion): renaming of parameter (choice of parameter name does not matter)

λx. E $\rightarrow_\alpha$ λz. (E[z/x]) provided **z** is not free in **E**
e.g., **λx. x x** is the same as **λz. z z**

β rule (β-reduction): function application (substitutes argument for parameter)

(λx.E) M $\rightarrow_\beta$ E[M/x]
Note: **E[M/x]** as defined on previous slide!
e.g., **(λx. x) z $\rightarrow_\beta$ z**

Exercise

Reduce

1. (λx. x) y → ? y

2. (λx. x) (λy. y) → ? λy.y

3. (λx.λy.λz. x z (y z)) (λu. u) (λv. v) → ?

$(\lambda y.\lambda z.(\lambda u.u)\, z\, (y\, z))\, (\lambda v.v) \rightarrow$

$\lambda z.(\lambda u.u)\, z\, ((\lambda v.v)\, z) \rightarrow$

$\lambda z.\, z\, ((\lambda v.v)\, z) \rightarrow \lambda z.\, z\, z$

Reductions

An expression **(λx.E) M** is called a redex (for reducible expression)

An expression is in normal form if it cannot be β-reduced

The normal form is the meaning of the term, the “answer”

Definitions of Normal Form

- Normal form (NF): a term without redexes
- Head normal form (HNF)
  - **x** is in HNF
  - **(λx. E)** is in HNF if **E** is in HNF
  - **(x E_1 E_2 … E_n)** is in HNF
- Weak head normal form (WHNF)
  - **x** is in WHNF
  - **(λx. E)** is in WHNF
  - **(x E_1 E_2 … E_n)** is in WHNF

Exercise

1. **λz. z z** is in NF, HNF, or WHNF?
2. **(λz. z z) (λx. x)** is in? *Neither*
3. **λx.λy.λz. x z (y (λu. u))** is in? *NF, HNF, WHNF*

Application expression.

4. $\underbrace{\textbf{(λx.λy. x)}}_{E_1}\ \underbrace{\textbf{z}}_{E_2}\ \underbrace{\textbf{((λx. z x) (λx. z x))}}_{E_3}$ is in? *Neither.*
5. **z ((λx. z x) (λx. z x))** is in? *HNF and WHNF.*
6. **(λz.(λx.λy. x) z ((λx. z x) (λx. z x)))** is in? *Only WHNF.*

An expression with no free variables is called combinator. pair, fst, snd are combinators.

pair = λx.λy.λf. f x y

fst = λf. f (λx.λy. x) **snd = λf. f (λx.λy. y)**

What is **fst (pair a b)**?

→ (λf. f (λx.λy. x)) (pair a b)

→ (pair a b) (λx.λy. x)

→ ((λx.λy.λf. f x y) a b) (λx.λy. x)

→ (λf. f a b) (λx.λy. x)

→ (λx.λy. x) a b

→ a

Reduction Strategy

- Let us look at **(λx.λy.λz. x z (y z)) (λu. u) (λv. v)**
- Actually, there are several "reduction paths":

Applicative:

$(\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z))(\lambda u.u)(\lambda v.v) \to$

$(\lambda y.\lambda z.\ (\lambda u.u)\ z\ (y\ z))(\lambda v.v) \to (\lambda y.\lambda z.\ z\ (y\ z))\ (\lambda v.v) \to$

$\lambda z.\ z\ ((\lambda v.v)\ z) \to \boxed{\lambda z.\ z\ z}$

Normal:

SAME ANSWER

$(\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z))\ (\lambda u.u)\ (\lambda v.v) \to$

$(\lambda y.\lambda z.\ (\lambda u.u)\ z\ (y\ z))\ (\lambda v.v) \to$

$\lambda z.\ (\lambda u.u)\ z\ ((\lambda v.v)\ z) \to \lambda z.\ z\ ((\lambda v.v)\ z) \to \boxed{\lambda z.\ z\ z}$

A reduction strategy (also called evaluation order) is a strategy for choosing redexes

How do we arrive at the normal form (answer)?

Applicative order reduction chooses the leftmost-innermost redex in an expression E: THIS ONE REDEX

In the sense that it contains no nested redexes

Also referred to as call-by-value reduction

Normal order reduction chooses the leftmost-outermost redex in an expression E: THIS REDEX

In the sense that it is not enclosed in a redex

Also referred to as call-by-name reduction

21

21

Exercises

Evaluate **(λx. x x) ((λy. y) (λz. z))** using applicative order reduction:

Evaluate **(λx. x x) ((λy. y) (λz. z))** using normal order reduction:

22

22

Exercise

Evaluate **(λx.λy. x y) ((λz. z) w)** using applicative order reduction:

Evaluate **(λx.λy. x y) ((λz. z) w)** using normal order reduction:

Programming in Haskell, A Milanova

23

23

Let Expressions

Adding one more term, the let-binding, for the purpose of studying Hindley Milner

E ::= x | (λx. E_1) | (E_1 E_2) | let x = E_1 in E_2

A λ-expression is one of

- Variable: **x**
- Abstraction (i.e., function definition): **λx. E_1**
- Application: **E_1 E_2**
- Let expression: **let x = E_1 in E_2**

Programming in Haskell, A Milanova

24

24

let in Haskell is a letrec allowing for general recursion

let x = E_1 in E_2

```
let
  plus = \x y -> if y==0 then x else plus (x+1) (y-1)
in
  plus 2 3
```

```
let
    even = \x -> if x==0 then True else odd (x-1)
    odd = \x -> if x==0 then False else even (x-1)
in
  even 100
```


25

Outline

- Pure lambda calculus, a review
  - Syntax and semantics
  - Free and bound variables
  - Rules (alpha rule, beta rule)
  - Normal forms
  - Reduction strategies
- Lazy evaluation in Haskell

Normal : E : _√_ _ _

Applicative : E : =√= _ _

TWO PARAMETERS:
1. REDUCTION STRATEGY
NORMAL or APPLICATIVE
2. NORMAL FORM
NF, HNF, WHNF

26

Back to Haskell

Expressions in Haskell are evaluated using lazy evaluation. What are its benefits?

- Avoids doing unnecessary evaluation;
- Ensures termination whenever possible;
- Supports programming with infinite lists;
- Allows programs to be more modular.

27

Evaluating Expressions

```
square n = n * n
```

Example:

```
  square (1+2)
=
  square 3
=
  3 * 3
=
  9
```

Apply + first.

I.e., applicative order reduction.

28

Or this:

```
  square (1+2)
= (1+2) * (1+2)
= 3 * (1+2)
= 3 * 3
= 9
```

Apply square first.

I.e., normal order reduction.

Church-Rosser theorem: Any way of evaluating the same expression will give the same result, provided it terminates (i.e., normal form exists).

Termination

```
infinity = 1 + infinity
```

> infinity

Example:

```
  fst (0, infinity)
= fst (0, 1 + infinity)
= fst (0, 1 + (1 + infinity))
=
  ⋮
```

Applicative order.


```
  fst (0, infinity)
= 0
```

Normal order evaluation.

Note:

- Outermost evaluation may give a result when innermost evaluation fails to terminate
- If any evaluation sequence terminates, then so does outermost, with the same result

Number of Reductions

Applicative order:

```
  square (1+2)
= square 3
= 3 * 3
= 9
```

3 steps.

Normal order:

```
  square (1+2)
= (1+2) * (1+2)
= 3 * (1+2)
= 3 * 3
= 9
```

4 steps.

Note:

- The outmost version is inefficient, because the argument 1+2 is duplicated when square is applied and is hence evaluated twice
- Due to such duplication, outermost evaluation may require more steps than innermost
- This problem can easily be avoided by using pointers to indicate sharing of arguments

Example:

```
square (1+2)

=

*        1+2

=

*        3

=

9
```

Shared argument evaluated once.

This gives a new evaluation strategy:

lazy evaluation = outermost evaluation + sharing of arguments

Note:

- Lazy evaluation ensures termination whenever possible, but never requires more steps than innermost evaluation and sometimes fewer
- Strategy is known as call-by-need. Haskell's evaluation strategy

Infinite Lists

Generatoren

itertools

```
gen x = x : gen x
```

Example:

```
  gen 1
= 1 : gen 1
= 1 : (1 : gen 1)
= 1 : (1 : (1 : gen 1))
= ⋮
```

An infinite list of ones.

Applicative order:

```
head (gen 1)
=
head (1:gen 1)
=
head (1:(1:gen 1))
=
⋮
```

Does not terminate.

Normal order:

```
head (gen 1)
=
head (1:gen 1)
=
1
```

Terminates in 2 steps!

Note:

- In the lazy case, only the first element of ones is produced, as the rest are not required
- In general, with lazy evaluation expressions are only evaluated as much as required by the context in which they are used
  driven by pattern match
- Hence, gen x is really a potentially infinite list

gen x = x : gen (x-1)

Modular Programming

[100,99..0]

[100,99..] is infinite

Lazy evaluation allows us to make programs more modular by separating control from data

```
> take 5 (gen 1)
[1,1,1,1,1]
```

repeat n v
| n == 0 = []
| otherwise = v : repeat (n-1) v

The data part (gen 1) is only evaluated as much as required by the control part take 5.

Without using lazy evaluation the control and data parts would need to be combined into one:

```
replicate :: Int → a → [a]
replicate 0 _ = []
replicate n x = x : replicate (n-1) x
```

Example:

```
> replicate 5 1
[1,1,1,1,1]
```

Generating Primes

To generate the infinite sequence of primes:

1. Write down the infinite sequence 2, 3, 4, …;
2. Mark the first number p as being prime;
3. Delete all multiples of p from the sequence;
4. Return to the second step.

Sieve of Eratosthenes.

2	3	4	5	6	7	8	9	10	11	12	…
	3		5		7		9		11		…
			5		7				11		…
					7				11		…
									11		…

This idea can be directly translated into a program that generates the infinite list of primes!

```
primes :: [Int]
primes = sieve [2..]
```

```
sieve :: [Int] → [Int]
sieve (p:xs) = p : sieve [x | x ← xs, x `mod` p ≠ 0]
```


Examples:

```
> primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,…
```

```
> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

```
> takeWhile (< 10) primes
[2,3,5,7]
```

Exercise

We can also use primes to generate an (infinite?) list of twin primes that differ by precisely two.

```
twin :: (Int,Int) → Bool
twin (x,y) =  x+2 == y
```

```
twins :: [(Int,Int)]
twins = filter twin (zip primes (tail primes))
```

```
> twins
[(3,5),(5,7),(11,13),(17,19),(29,31),...
```

2, 3, 5, 7, 11, 13

3, 5, 7, 11, 13

Programming in Haskell, slide due to G. Hutton 45

Exercise

fibs = 0 : 1 : fibs'

1 : 1 : fibs'

0 : 1 : 1 : 2

1 : 1 : 2

(1) The Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

starts with 0 and 1, with each further number being the sum of the previous two. Using a list comprehension, define an expression

```
fibs :: [Integer]
fibs = 0:1:fibs' where
 fibs' = ~~[ x+y | x <- fibs, y <- tail fibs ]~~
 [ x+y | (x,y) <- zip fibs (tail fibs)]
```

that generates this infinite sequence.

Programming in Haskell, slide due to G. Hutton 46

Pattern Matching

[(x,y) | x <- [1,2,3], y <- [4,5]]

= [(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]

Pattern matching drives evaluation

```
f1 :: Maybe a -> [Maybe a]
f1 m = [m,m]

f2 :: Maybe a -> [a]
f2 Nothing = []
f2 (Just x) = [x]
```

zip [1,2,3] [4,5]

→ [(1,4), (2,5)]

f1's argument remains completely uneveluated

f2 e must first evaluate argument e because result of f2 depends on the shape of e

Thunks are evaluated only as much as needed. E.g., safeHead [3^10, 5] does not evaluate 3^10

Programming in Haskell, A Milanova (example due to Brent Yorgey, Haskell.org) 47


```
repeat :: a -> [a]
repeat x = x : repeat x

take :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

```
take 2 (repeat 1)
{- match 2, n <= 0 fails, goes to next match -}
take 2 ( 1 : repeat 1 ) {- matches with (x:xs) -}
1 : take (2-1) (repeat 1)
1 : take 1 (repeat 1)
1 : take 1 ( 1 : repeat 1)
1 : 1 : take (1-1) (repeat 1)
1 : 1 : take 0 (repeat 1) {- match n <= 0, return [] -}
1 : 1 : []
```

Programming in Haskell, A Milanova (example due to Brent Yorgey, Haskell.org) 48

Understand space usage. Remember foldl:

```
foldl (+) 0 [1,2,3]
= foldl (+) (0+1) [2,3]
= foldl (+) ((0+1)+2) [3]
= foldl (+) (((0+1)+2)+3) []
= (((0+1)+2)+3)
= ((1+2)+3)
= (3+3)
= 6

> foldl (+) 0 [1..1000000]
500000500000
(0.27 secs, 161,298,016
bytes)
```

```
foldl’ (+) 0 [1,2,3]
= foldl’ (+) (0+1) [2,3]
= foldl’ (+) 1 [2,3]
= foldl’ (+) (1+2) [3]
= foldl’ (+) 3 [3]
= foldl’ (+) (3+3) []
= foldl’ (+) 6 []
= 6

> foldl’ (+) 0 [1..1000000]
500000500000
(0.03 secs, 88,071,264
bytes)
```


Another problem is that evaluating (((0+1)+2)+3) requires pushing 3, then 2, etc. on a stack and then unwinding the stack while adding along the way. This adds to space usage

```
…
= (((0+1)+2)+3)
= ((1+2)+3)
= (3+3)
= 6

> foldl (+) 0 [1..1000000]
500000500000
(0.27 secs, 161,298,016
bytes)
```


Strict Evaluation

One can force strict evaluation with bang patterns

```
f1 :: a -> Bool
f1 x = True
```

```
f1’ :: a -> Bool
f1’ !x = True
```

```
> f1 inifinty
True
> f1’ infinity
                    WHNF
> f1’ (\x -> fst (x, infinity))
True
> f1’ (fst (0, infinity))
True
```

```
foldl op i xs
foldl op !i [] = i
foldl op !i (x:xs) =
  foldl op (op i x) xs
```


Short-circuiting Operations

In Haskell short-circuiting is natural

```
&& :: Bool -> Bool -> Bool
True && x = x
False && _ = False
```

```
&&! :: Bool -> Bool -> Bool
True &&! True = True
False &&! False = False
False &&! True = False
False &&! False = False
```

```
> False && (34^9784346 > 34987345)
False
(0.01 secs, 68,040 bytes)
> False &&! (34^9784346 > 34987345)
False
(0.32 secs, 18,142,296 bytes)
```

```
False && (head [] == 'x')
False &&! (head [] == 'x')
```

Arrays

import Data.Array

```
array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

A one-dimensional array of squares: A[i] = i^2:

```
squares   =  array (1,100) [(i, i*i) | i <- [1..100]]
```

A two-dimensional array: A[i,j] = i + j:

```
sums = array ((1,1),(100,100)) $
      [((i,j), i+j) | i <- [1..100], j <- [1..100] ]
```


Dynamic Programming

In Haskell dynamic programming is natural

```
import Data.Array
knapsack01 :: [Double] -- values
           -> [Integer] -- nonnegative weights
           -> Integer -- knapsack size
           -> Double -- max possible value
knapsack01 vs ws maxW = m!(numItems-1, maxW)
   where numItems = length vs
         m = array ((-1,0), (numItems-1, maxW)) $
               [((-1,w), 0) | w <- [0 .. maxW]] ++
               [((i,0), 0) | i <- [0 .. numItems-1]] ++
               [((i,w), best)
                   | i <- [0 .. numItems-1]
                   , w <- [1 .. maxW]
                   , let best
                          | ws!!i > w = m!(i-1, w)
                          | otherwise = max (m!(i-1, w))
                                            (m!(i-1, w - ws!!i) + vs!!i)]
```


Exercise

Define longest common subsequence

```
import Data.Array
lcs :: [a] -- sequence a
    -> [b] -- sequence b
    -> Integer -- length of lcs of a and b
lcs seqa seqb = m!(la - 1, lb - 1)
   where …
```


Lab

- Work on HW5
- Work on generic monadic functions (download Lecture6’.hs) and monoids and foldables from Lecture7
- Work on HW6