Lambda Calculus

CS242

Lecture 4

Review

- Reduction order
  - Where should the next reduction be performed?
  - Normal order: always choose the leftmost, outermost reduction
- Confluence
  - If a computation terminates, the result is always the same regardless of the evaluation order used
- Primitive recursion/array programming
  - Use whole datatype operations for concise, loop-free programs

History

- The lambda calculus was one of several computational systems defined by mathematicians to probe the foundations of logic
  - Others: combinator calculus, Turing machines
- Lambda calculus was introduced by Alonzo Church in the 1930's
  - Originally used to establish the existence of an undecidable problem

A Language of Functions

- Like SKI calculus, lambda calculus focuses exclusively on functions
- Unlike SKI, lambda calculus has a notion of variable

$e \rightarrow x \mid \lambda x.e \mid e\ e \mid (e)$

In words, a lambda expression is a

variable x,

an *abstraction* (a function definition) $\lambda x.e$, or

an *application* (a function call) $e_1\ e_2$

Intuition

A function $\lambda x.e$ is a function definition just like

def f(x) = e

Two differences

$\lambda x.e$ is an anonymous function – it doesn’t have a name like “f”

$\lambda x.e$ is a value – it can be a function argument or result

Association

Rule: The body of a lambda abstraction extends as far right as possible.

to the end of the expression or an unmatched right paren

$\lambda x.x\ \lambda y.y = \lambda x.(x\ \lambda y.y)$

$\lambda x.(\lambda y.\lambda z.y\ z)\ x$ is different from $\lambda x.\lambda y.\lambda z.y\ z\ x = \lambda x.\lambda y.\lambda z.(y\ z\ x)$

Rule: Application associates to the left

So $f\ x\ y\ z = ((f\ x)\ y)\ z$

Computation Rule

$$(\lambda x.e_1)\ e_2 \rightarrow e_1\ [x := e_2]$$

In words: In a function call, the *formal parameter* x is replaced by the *actual argument* e_2 in the *body* of the function e_1.

This is called *beta reduction.*

Examples

- The identity function I: $\lambda x.x$

- The constant function K: $\lambda z.\lambda y.z$

$(\lambda x.x)\ (\lambda z.\lambda y.z) \rightarrow x\ [x := \lambda z.\lambda y.z] = \lambda z.\lambda y.z$

$((\lambda z.\lambda y.\ z)\ (\lambda x.x))\ (\lambda a.\lambda b.a) \rightarrow (\lambda y.\ (\lambda x.x))\ (\lambda a.\lambda b.a) \rightarrow \lambda x.x$

Substitution

- Beta-reduction is the workhorse rule in the lambda calculus
  - But it relies on substitution

$x\,[x := e] \;=\; e$

$y\,[x := e] \;=\; y$

$(e_1\ e_2)\,[x := e] = (e_1\,[x := e])\ (e_2\,[x := e])$

$(\lambda x.e_1)\,[x := e] = \lambda x.e_1$

$(\lambda y.e_1)\,[x := e] = \lambda y.(e_1\,[x := e])$ if $x \neq y$ and y does not appear free in e

Huh?

Why do we need this complicated rule?

$(\lambda y.e_1)\ [x := e] = \lambda y.(e_1\ [x := e])$ if $x \neq y$ and y does not appear free in e

Consider

$(\lambda y.x)\ [\ x := y\ ]$

We don't want the answer to be $\lambda y.y$!

Free Variables

The *free variables* of an expression are the variables not bound in an abstraction.

$FV(x) = \{ x \}$

$FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$

$FV(\lambda x.e) = FV(e) - \{ x \}$

Substitution Revisited

$x\ [x := e] = e$

$y\ [x := e] = y$

$(e_1\ e_2)\ [x := e] = (e_1\ [x := e])\ (e_2\ [x := e])$

$(\lambda x.e_1)\ [x := e] = \lambda x.e_1$

$(\lambda y.e_1)\ [x := e] = \lambda y.(e_1\ [x := e])$ if $x \neq y$ and $y \notin FV(e)$

But Substitution Should Always Work ...

- Intuitively, the bound variable name in an abstraction doesn't matter
  - $\lambda x.x$ is as good as $\lambda y.y$

- We can rename bound variables to avoid name collisions:

$(\lambda y.e_1)\ [x := e] = \lambda z.((e_1[y := z])\ [x := e])$ if $x \neq y$ and z is a fresh name

(*fresh* means not occurring in e_1 or e)

Revisiting Our Substitution Example ...

$(\lambda y.x)\ [\ x := y\ ]\quad =$

$(\lambda z.x)\ [\ x := y\ ] =$

$(\lambda z.y)$

Rules Again

- Renaming of bound variables is called *alpha conversion*

- Presentations of lambda calculus often include alpha conversion as a separate rule

- A third rule, *eta-conversion*, is also part of the lambda calculus but is not needed for computation:

$$e = \lambda x.e\ x \qquad x \notin FV(e)$$

Summary

Lambda calculus has three rules:

- *Beta reduction* $(\lambda x.e_1)\ e_2 \rightarrow e_1\ [x := e_2]$
- *Alpha conversion* $\lambda x.e = \lambda z.e\ [\ x := z]$ where z is fresh
- *Eta conversion* $\lambda x.e\ x = e \quad x \notin FV(e)$

Lambda calculus is often presented emphasizing only beta reduction, with alpha conversion assumed to be done where needed to avoid capture of free variables ("capture-avoiding renaming"). Eta conversion is used mostly in proofs of logical properties, not in direct computation.

Summary

- Lambda calculus is a language of higher-order functions

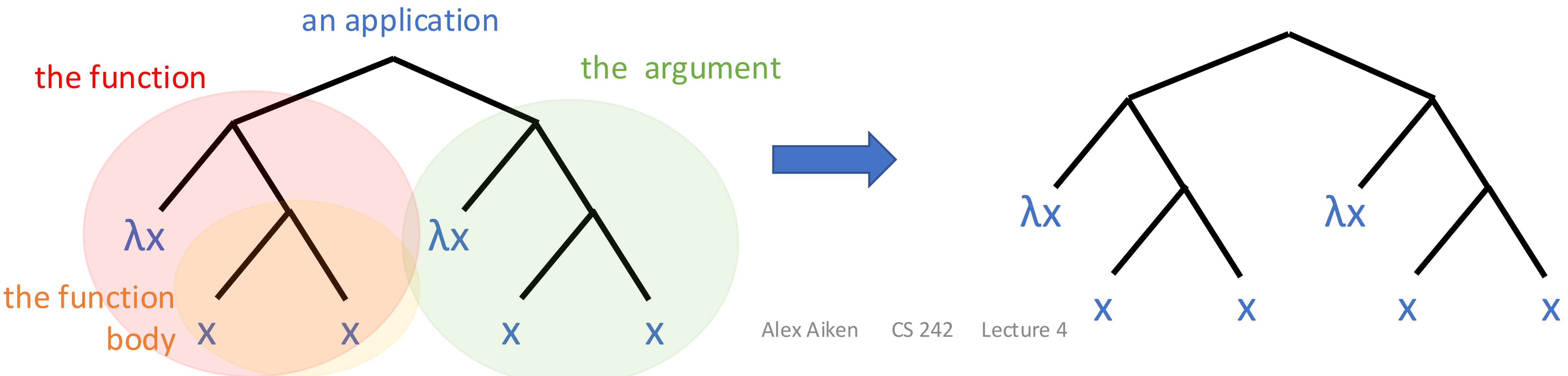
- Looks more familiar than SKI
  - At least it has variables for function arguments!

- But there is a cost
  - Defining how an expression is substituted for a variable is a little tricky
  - Need to be careful not to inadvertently cause clashes of different variables with the same name
  - Requires renaming variables in general

Example

$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \rightarrow\ x\ x\ [x := \lambda x.\ x\ x] = (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

- An example of a non-terminating expression
  - Reduces to itself in one step, so can always be reduced



Recursion

As with SKI, producing true recursion is just slightly more involved:

$Y = \lambda f.(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f(x\ x))$

$Y\ g\ a = (\lambda f.(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f(x\ x)))\ g\ a \rightarrow$
$(\lambda x.\ g\ (x\ x))\ (\lambda x.\ g(x\ x))\ a \rightarrow$
$g((\lambda x.\ g(x\ x))\ (\lambda x.\ g(x\ x)))\ a \rightarrow$
$g(g((\lambda x.\ g(x\ x))\ (\lambda x.\ g(x\ x))))\ a \rightarrow$
…

Booleans

- As with SKI, represent true (false) by a function that given two arguments picks the first (second)

- True = K = $\lambda x.\lambda y.x$
- False = $\lambda x.\lambda y.y$

- Example $(\lambda x.\lambda y.y)\ w\ z \rightarrow (\lambda y.y)\ z \rightarrow z$

Equations and Functions

- We could also start with equations for True and False

  True x y = x

  False x y = y

- Now we need to convert these to lambda terms
  - Much like the abstraction algorithm we used for SKI

- But this procedure is *easy* in lambda calculus:
  - Each variable on the left side becomes a lambda abstraction on the right side
  - In the same order

- True = $\lambda x.\lambda y.x$
- False = $\lambda x.\lambda y.y$

Boolean Operations

- Note that our definitions of True and False are combinators
  - They have no free variables
  - So we can just reuse the SKI encoding of the Boolean operations

- Let B be a Boolean
- not(B) = B False True
- B1 or B2 = B1 True B2
- B1 and B2 = B1 B2 False

Pairs

pair x y z = z x y

fst x y = x

snd x y = y

pair = λx.λy.λz. z x y

fst = λx.λy.x

snd = λx.λy.y

pair True False first =

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

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

(λz. z (λx.λy.x) (λx.λy.y)) (λx.λy.x)

(λx.λy.x) (λx.λy.x) (λx.λy.y)

(λy.λx.λy.x) (λx.λy.y)

λx.λy.x =

True

Natural Numbers

- n applies its first argument n times to its second argument

$n\ f\ x = f^n(x)$

$0\ f\ x = x$ so $0 = \lambda f.\lambda x.x$

$succ\ n\ f\ x = f\ (n\ f\ x)$ $succ = \lambda n.\lambda f.\lambda x.\ f\ (n\ f\ x)$

Factorial

one = succ 0
add = λm.λn. m succ n
mul = λm.λn. m (add n) 0

pair = λa.λb.λf. f a b
fst = λx.λy.x
snd = λx.λy.y

p = λp. pair (mul (p fst) (p snd)) (succ (p snd))
! = λn.(n p (pair one one) fst)

And The Rest: Some Lambda Calculus Topics

- The lambda calculus is extremely well-studied
  - More studied than combinator systems

- We'll touch on a few highlights:
  - Algebraic data types
  - General vs. primitive recursion
  - Confluence
  - Call-by-name vs. call-by-value
  - Implementing lambda calculus using SKI

Algebraic Data Types

- An algebraic data type is a data type that is a union of multiple cases
  - Each case is a function called a *constructor* with a fixed number of arguments
  - Algebraic data types can be recursively defined

- Schematically:

Type T=

constructor$_1$ Type$_{11}$ Type$_{12}$ … Type$_{1n}$ |

constructor$_2$ Type$_{21}$ Type$_{22}$ … Type$_{2m}$ |

… more constructors …

Comments:

The type arguments can be Bool, Int, Char, T itself or other ADTs

The data type is “algebraic” because the constructor simply packages up the arguments

The constructor functions as a “tag” naming which case of the ADT is being used

A corresponding *deconstructor* recovers the constructor arguments for computing on the ADT

Natural Numbers, Reprise

- The natural numbers are an example of an algebraic data type

```
Type Nat = succ Nat |
           0
```

- Two constructors
  - succ of arity 1
  - 0 of arity 0 (a constant with no arguments)

Lists of Natural Numbers

Type List = nil |

cons Nat List

- Two constructors
  - nil of arity 0 (a constant with no arguments)
  - cons of arity 2

Binary Trees of Natural Numbers

Type Tree = leaf Nat |
branch Tree Tree

- Two constructors
  - leaf of arity 1
  - branch of arity 2

Encoding Algebraic Types in Lambda Calculus

Consider an algebraic data type T with n constructors

Let the ith constructor C_i have k arguments

The constructor and destructor for C_i can be implemented by one term:

$$\lambda a_1.\ \lambda a_2.\ \ldots\ \lambda a_k.\ \lambda f_1.\ \lambda f_2.\ \ldots\ \lambda f_n.\ f_i\ a_1\ a_2\ \ldots\ a_k$$

The first k arguments are the constructor part: We take k arguments to build an element of T.

The rest is an element of the ADT. Every element of type T takes one function for each constructor of T.

An element of the ith constructor applies the ith function to the constructor's k arguments.

Not shown: Arguments of type T are recursively passed the n functions (see examples)

A Simple Example: Pairs of Natural Numbers

Type Pair = P Nat Nat

Implementation:

λa.λb.λf. f a b

- Two arguments to build an element of constructor P
- Only one constructor, so the destructor only takes one function, which it applies to the two arguments

Natural Numbers, Reprise

Type Nat = succ Nat |
0

0 = λf.λx.x

- 0 has no arguments – the "constructor" is a constant value
- Nat has two constructors, so the destructor always takes two functions, f for the succ case and x for the 0 case. Since 0 has no arguments we just return x

Natural Numbers, Reprise

Type Nat = succ Nat |
                0

succ = λn.λf.λx. f (n f x)

- succ has one argument n
- The destructor takes two functions, f for succ and x for 0
- Since natural numbers are recursively defined (n is of type Nat), we apply f to the result of recursively computing n f x

Lists of Natural Numbers

Type List = nil |
cons Nat List

cons = λh.λt.λx.λf. f h (t x f)

nil = λx.λf.x

Summing a List of Natural Numbers

```
# natural numbers
0 = λf.λx.x
succ = λn.λf.λx. f (n f x)

# lists
nil = λx.λf.x
cons = λh.λt.λx.λf. f h (t x f)

1 = succ 0
add = λm.λn. m succ n
sum = λl.l 0 add
test = sum (cons 1 (cons 0 (cons 0 nil)))
```

Intuition: How Does Recursion on ADTs Work?

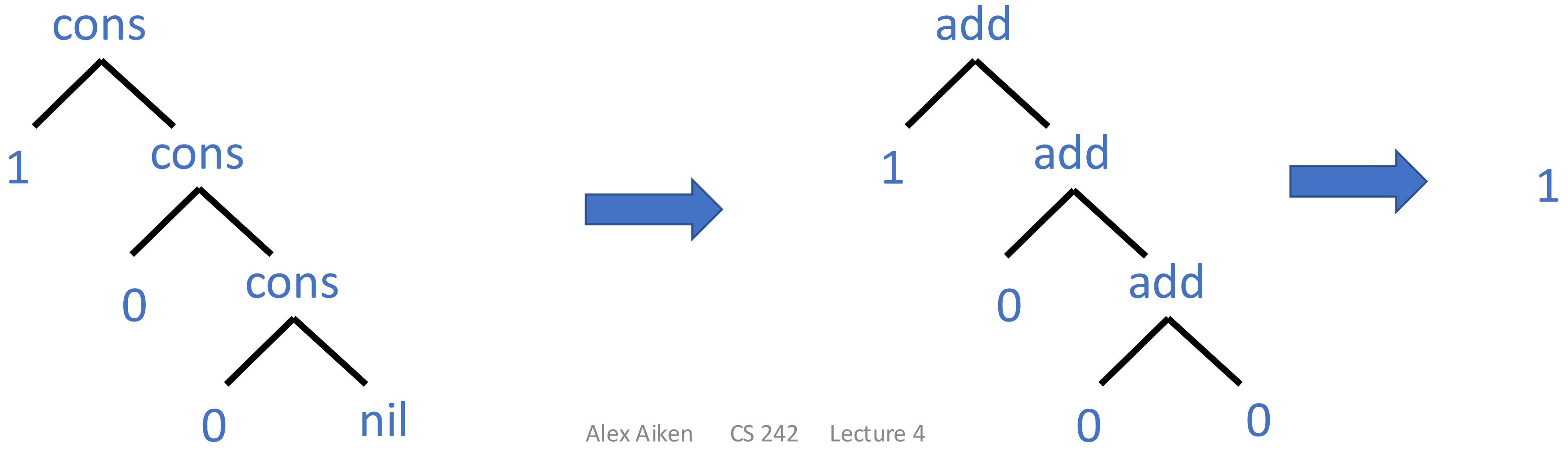
sum = λl.l 0 add

test = sum (cons 1 (cons 0 (cons 0 nil)))

So test = (λl.l 0 add) (cons 1 (cons 0 (cons 0 nil)))

Intuition: Replace the constructors with corresponding functions and evaluate the result!

λl.l 0 add



Primitive Recursion

- Primitive recursion is the difference between
  - for I = 1 to 10 do …
  - while (predicate(x)) do … something that modifies x ….
- In the first case the number of iterations is fixed when the loop starts
  - Termination is guaranteed!
- Many data structures lend themselves naturally to primitive recursion
  - Do something with every element of an array
  - Traverse a list
  - Iterate from 1 to n or n to 1
  - This pattern is captured in a general way in our definition of algebraic data types
- In general recursion, the decision of whether to loop depends on data computed within the loop
  - Sometimes general recursion is necessary – not everything can be written using primitive recursion
  - But general recursion is more complex – you need a separate termination argument to understand why your loop will eventually stop

Confluence

- The lambda calculus is confluent
  - The Church-Rosser theorem

- If $e_0 \to^* e_1$ and $e_0 \to^* e_2$, then there is an e_3 s.t. $e_1 \to^* e_3$ and $e_2 \to^* e_3$
  - Where we consider terms equivalent up to alpha conversion

- The proof is similar to the SKI proof
  - But not as short …

Reduction Order

Given a *redex* (λx.e) e' should we:

- Evaluate e' before performing the beta reduction? *call-by-value*
- Perform the beta reduction first? *call-by-name*

- Normal order (or lazy evaluation, or call-by-name) is the same as in SKI
  - Always reduce the leftmost, outermost redex

- In call-by-value (or eager evaluation), we first recursively evaluate the argument before reducing the function application
  - The strategy used in C, C++, python, Java – probably every language you have used

Does The Reduction Order Matter?

- Answer 1: It mostly doesn't matter, because of confluence

- Answer 2: For efficiency, call-by-value is better
  - Evaluate arguments one time

- Answer 3: For termination, call-by-name is better
  - Call-by-name is guaranteed to terminate, if termination is possible
  - Call-by-value may fail to terminate even if call-by-name terminates
  - Does not contradict confluence, which says there is *some* reduction sequence to reach a common term, not that a particular reduction strategy will reach it
  - Recall that primitive recursion trivially guarantees termination

Implementation

- There are many ways to implement lambda calculus
  - One method is to translate lambda terms to SKI combinators

- Recall the abstraction algorithm: $A(E,x)\ x = E$

- Observe that $\lambda x.e = A(E,x)$
  - And $A(E,x)$ is an SKI expression if e contains no lambda abstractions

- Consider a lambda expression e
  - Repeat until there are no lambda abstractions remaining
    - Replace an innermost lambda expression $\lambda x.e'$ in e by $A(e',x)$

Equivalences

- The following are all equivalent in computational power
  - SKI calculus
  - Lambda calculus
  - Turing machines

- Next time we will talk about typed lambda calculus, which is strictly less powerful.