CMSC330: The Lambda Calculus

Chris Kauffman

Last Updated:
Mon Oct 30 11:27:33 PM EDT 2023

Logistics

Reading

Types and Programming Languages, Ch 5 by Benjamin C. Pierce

- Accessible reference on Lambda Calculus
- Explores other topics of interest in theory of PLs

Lambda-Calculus and Combinators, an Introduction by Hindley and Seldin

- More technical but what would you expect from Hindley, co-inventor of type inference

Goals

☒ Wrap-up of Parsing / Evaluation

☒ Begin Survey of Lambda Calculus

☐ Encodings in Lambda Calculus

Assignments

- Project 5 up and running
- NFA to DFA conversion in OCaml
- P5 due 30-Oct
- In-Person Quiz 3 Moved to Next Week (Fri 03-Nov)

Announcements

- This is a rough week for UMD
- If you're feeling that roughness, actively find a way to smooth out,
  - Pause and Rest
  - Talk to your people, let them know how they are feeling
  - Look after your people, find out how they are feeling

Church-Turing Thesis

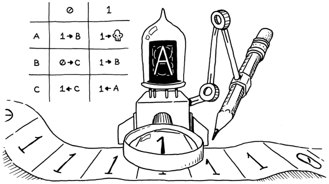
- Mathematicians wished to formalize the notion of an “algorithm”
- A variety of different models were proposed the best know of which are
  - The Lambda Calculus by Alonzo Church
  - Turing Machines by Alan Turing (originally called “a-machines”
- Work by Turing, Church, Stephen Kleene (ring any bells?) and others showed that these two models of algorithms (and other models such as one proposed by Kurt Gödel) are equivalent
- Result: **Church-Turing** Thesis, informally
  - “X is computable if one can build a Turing machine for it”
  - “X is computable if one can encode in the Lambda Calculus”

Turing Machines

Natural extension of Finite Automata which we studied earlier

- An infinite tape with cells, each containing a symbol (e.g. 1 or 0)
- A head positioned at one cell
- A finite set of states including a starting state
- A finite transition table of instructions like the one below

State/Tape	Head	Move	Next
A / 0	Write 1	L	A
A / 1	-	R	B
B / 0	-	R	A
B / 1	Write 0	L	C
...	...	...	...



Source: Crafting Interpreters

Turing machines appeal as smack of a mechanical device and most real computers directly derive from these ideas including machines Turing himself helped construct

The Lambda Calculi

What is usually called λ-calculus is a collection of several formal systems, based on a notation invented by Alonzo Church in the 1930s. They are designed to describe the most basic ways that operators or functions can be combined to form other operators.

In practice, each λ-system has a slightly different grammatical structure, depending on its intended use. Some have extra constant symbols, and most have built-in syntactic restrictions, for example type restrictions. But to begin with, it is best to avoid these complications; hence the system presented in this chapter will be the pure one, which is syntactically the simplest.

– Hindley and Seldin in "Lambda-Calculus and Combinators, an Introduction"

- We will focus on the Pure Lambda Calculus / Untyped Lambda Calculus
- OCaml follows the Simply Typed Lambda calculus more closely but that's beyond our scope here

The Untyped Lambda Calculus

The Grammar

- Described via a CFG
- Quite simple with only 3 parts

$T \to x$	Variable name	(1)
$T \to \lambda x.T$	Abstraction	(2)
$T \to T\,T$	Application	(3)

- Variable names are any lowercase letter $x, y, z, \ldots$
- (2) referred to as "lambda abstraction" in some cases

Examples of Derived Strings

Sometimes referred to as "Lambda Terms"

1. y
2. $\lambda x.x$
3. $\lambda y.z$
4. $x\,y$
5. $z\,z$
6. $x\,y\,z \equiv (x\,y)z$
7. $\lambda x.\lambda y.x\,y$
8. $x\,(\lambda y.xyz)$
9. $(\lambda x.\lambda y.x\,y)\,a\,b$
10. $(\lambda x.x\,x)(\lambda y.y\,y\,y)$

A Few Notes on Syntax

Application Associates Left

While the CFG given is technically ambiguous, Applications are always assumed to be Left Associative:

- $x\,y\,z \equiv (x\,y)\,z$
- $a\,b\,c\,d \equiv ((a\,b)\,c)\,d$

Note OCaml uses the same syntax and associativity for function application

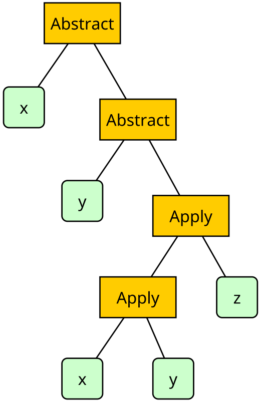
Abstraction Scope

Abstraction body is implicitly parenthesized

- $\lambda x.y\,x \equiv \lambda x.(y\,x)$
- $\lambda a.\lambda b.\lambda c.a\,b\,c \equiv$
  $\lambda a.(\lambda b.(\lambda c.((ab)c)))$

CFG Implies Syntax Tree

CFG implies tree structures in lambda terms
Ex: $\lambda x.\lambda y.x\,y\,z$



Bound and Free Variables, Combinators

- x is **bound** when it occurs as within the **body** of an Abstraction $\lambda x. \dots x \dots$
- The Abstraction is the **binder** of x: within its body, x will have a specific definition assigned it
- If a variable is not bound, it is **free**

Examples:

1. $\lambda x.xy$
   x is bound, y is free
2. $x\,(\lambda y.\lambda z.z\,y)$
   y, z are bound, x is free

Combinator: terms with no free variables / *closed terms*
The simplest Combinator is the **Identity Function**:

- $\lambda x.x$ (*Identity Function*)

The Pure Lambda Calculus is interesting because of combinators. . .

Exercise: Alpha Conversion

- Two terms that differ only in the names bound variables are considered identical if they only differ in the names of bound variables
- Examples:
  1. $\lambda x.x \equiv \lambda y.y$ *(Both Identity Function)*
  2. $\lambda x.\lambda y.x\, y \equiv \lambda q.\lambda r.q\, r$
- Lingo: "Terms are equal up to renaming of bound variables."
- Church dubbed this "Alpha-Conversion": consistently rename bound variables to reveal structural equivalence

Given the Abstract Syntax Tree for a Lambda Term, write an algorithm for consistent variable renaming

Answers: Alpha Conversion

- Initialize a counter i to 0 (e.g. start with Variable Zero)
- Perform a tree traversal (tree walk) on AST of lambda term
- Whenever an "Abstract" node is encountered
- Replace the bound variable x that appears with v_i
- Each later appearance of x is substituted with the **fresh variable** v_i
- Increment the counter so the next variable will be v_{i+1}: next variable name is again fresh

$$\lambda x.\lambda y.x\, y \equiv_{alpha} \lambda v_0.\lambda v_1.v_0\, v_1$$
$$\lambda q.\lambda q.q\, r \equiv_{alpha} \lambda v_0.\lambda v_1.v_0\, v_1$$

You'll likely have to code this algorithm in OCaml in an upcoming project

Semantics of Lambda Calculus

- Have describe parts of what Lambda Calculus **is** via grammar, definitions
- But what does it **do**?
  - What are its semantics?
  - How does on evaluate a lambda term?

There is only 1 operation: *Application* of functions and **reduction** of the resulting terms

- Application of variables (atoms) doesn't reduce further
  - $x\,y$: nothing to do, already in **normal form**
  - $q\,r\,s$: nothing to do
- Application of an Abstraction **substitutes** bound variables with their parameter in the abstraction body
  - $(\lambda x.x)\,y \Rightarrow y$
  - $(\lambda z.z\,z)\,w \Rightarrow w\,w$
- Notation for Substitution: $(\lambda x.t_1)\,t_2 \Rightarrow [x \mapsto t_2]\,t_1$
  Spoken "substitute t_2 for x in t_1"

Exercise: Beta-Reduction and Normal Forms

- Reducing / Evaluating lambda terms is referred to as **Beta-Reduction**
- In many cases it terminates: reach a stage where no further reductions are possible, referred to as a **Normal Form** or **Beta-Normal Form**
- Try reducing the following terms to normal form

$$(\lambda x.x)\,a \Rightarrow ? \qquad (A)$$

$$(\lambda x.x)\,(\lambda y.y) \Rightarrow ? \qquad (B)$$

$$(((\lambda x.\lambda y.y\,x\,y)\,a)\,b) \Rightarrow ? \qquad (C)$$

$$(\lambda x.x(\lambda y.y))\,(u\,r) \Rightarrow ? \qquad (D)$$

$$\lambda x.(x\,((\lambda y.y)\,a)) \Rightarrow ? \qquad (E)$$

$$((\lambda x.x)\,a)\,((\lambda x.x)\,b) \Rightarrow ? \qquad (F)$$

Answers: Beta-Reduction and Normal Forms

Normal Forms for left-hand side shown right of $\Rightarrow$ arrows

$$\begin{aligned}
(\lambda x.x)\, a &\Rightarrow a && (A)\\
(\lambda x.x)\, (\lambda y.y) &\Rightarrow (\lambda y.y) && (B)\\
(\lambda x.\lambda y.y\, x\, y)\, a\, b &\Rightarrow (\lambda y.y\, a\, y)\, b && (C)\\
&\Rightarrow b\, a\, b\\
(\lambda x.x\, (\lambda y.y))\, (u\, r) &\Rightarrow (u\, r)\, (\lambda y.y) && (D)\\
\lambda x.(x\, ((\lambda y.y)\, a)) &\Rightarrow \lambda x.(x\, a) && (E)\\
((\lambda x.x)\, a)\, ((\lambda x.x)\, b) &\Rightarrow a\, b && (F)
\end{aligned}$$

(E) might feel a bit strange: is it really okay to Apply the Identity function $\lambda y.y$ inside another abstraction?
(F) might cause you to wonder whether to evaluate identity on a or b first and whether it matters.
That depends on your evaluation strategy...

Evaluation Strategies

- Beta Reduction indicates *What* to do (reduce application terms via substitution)
- Does not specify *How* to do it: e.g. the order substitutions should take place
- May see following technical terminology:
  - **Big Step Semantics**: one "step" entirely reduces a lambda term to normal form, focus on results
  - **Small Step Semantics**: one "step" only reduces by reducing a single function Application, focus on process
- **Evaluation Strategies** describe which function application to "fire" to take a step towards normal form AND specify how far to go towards a normal form

Eager vs Lazy Evaluation Strategies

Lazy Evaluation

- *Call by Name*
- Reduce Leftmost / Outermost Application first
- Abstractions that that are not applied do not reduce

In short: *Substitute entire argument first* into abstractions

Eager / Strict Evaluation

- *Call by Value*
- Evaluate "argument" to Applications first by reducing them to their normal form
- Then perform substitution within Abstractions
- Abstractions that that are not applied do not reduce

In short: *Reduce argument first* before subbing into abstractions

Exercise: Examples of Eager vs Lazy Eval

Lazy / Non-strict Evaluation

In short: *Substitute entire argument first* into abstractions

$$(\lambda \underline{z}.z)\,\underline{((\lambda y.y)\,x)} \qquad \text{(A)}$$
$$\Rightarrow (\lambda \underline{y}.y)\,\underline{x}$$
$$\Rightarrow x$$

$$(\lambda x.\lambda y.(y\,x))\,((\lambda z.z)\,w) \qquad \text{(B)}$$
$$\Rightarrow ???$$
$$\Rightarrow \lambda y.(y\,w)$$

$$(\lambda x.x\,w)\,((\lambda y.y)\,(\lambda z.(\lambda u.u)\,z)) \qquad \text{(C)}$$
$$\Rightarrow ???$$
$$\Rightarrow ???$$
$$\Rightarrow w$$

Eager / Strict Evaluation

In short: *Reduce argument first* before subbing into abstractions

$$(\lambda z.z)\,((\lambda \underline{y}.y)\,\underline{x}) \qquad \text{(A)}$$
$$\Rightarrow (\lambda \underline{z}.z)\,\underline{x}$$
$$\Rightarrow x$$

$$(\lambda x.\lambda y.(y\,x))\,((\lambda z.z)\,w) \qquad \text{(B)}$$
$$\Rightarrow ???$$
$$\Rightarrow \lambda y.(y\,w)$$

$$(\lambda x.x\,w)\,((\lambda y.y)\,(\lambda z.(\lambda u.u)\,z)) \qquad \text{(C)}$$
$$\Rightarrow ???$$
$$\Rightarrow ???$$
$$\Rightarrow ???$$
$$\Rightarrow w$$

Answers: Examples of Eager vs Lazy Eval

Lazy / Non-strict Evaluation

In short: *Substitute entire argument first* into abstractions

$$(\lambda \underline{z}.z)\ \underline{((\lambda y.y)\ x)} \qquad \text{(A)}$$
$$\Rightarrow (\lambda \underline{y}.y)\ \underline{x}$$
$$\Rightarrow x$$

$$(\lambda \underline{x}.\lambda y.(y\ x))\ \underline{((\lambda z.z)\ w)} \qquad \text{(B)}$$
$$\Rightarrow \lambda y.(y\ ((\lambda \underline{z}.z)\ \underline{w}))$$
$$\Rightarrow \lambda y.(y\ w)$$

$$(\lambda \underline{x}.x\ w)\ \underline{((\lambda y.y)\ (\lambda z.(\lambda u.u)\ z))} \qquad \text{(C)}$$
$$\Rightarrow ((\lambda \underline{y}.y)\underline{(\lambda z.(\lambda u.u) z)})w$$
$$\Rightarrow (\lambda \underline{u}.u)\underline{w}$$
$$\Rightarrow w$$

Eager / Strict Evaluation

In short: *Reduce argument first* before subbing into abstractions

$$(\lambda z.z)\ ((\lambda \underline{y}.y)\ \underline{x}) \qquad \text{(A)}$$
$$\Rightarrow (\lambda \underline{z}.z)\ \underline{x}$$
$$\Rightarrow x$$

$$(\lambda x.\lambda y.(y\ x))\ ((\lambda \underline{z}.z)\ \underline{w}) \qquad \text{(B)}$$
$$\Rightarrow (\lambda \underline{x}.\lambda y.(y\ x))\ \underline{w}$$
$$\Rightarrow \lambda y.(y\ w)$$

$$(\lambda x.x\ w)\ ((\lambda \underline{y}.y)\ \underline{(\lambda z.(\lambda u.u)\ z)}) \qquad \text{(C)}$$
$$\Rightarrow (\lambda \underline{x}.x\ w)\ \underline{(\lambda z.(\lambda u.u)\ z)}$$
$$\Rightarrow (\lambda \underline{z}.(\lambda u.u)\ z)\ \underline{w}$$
$$\Rightarrow (\lambda \underline{u}.u)\ \underline{w}$$
$$\Rightarrow w$$

Totally Legit Questions 1/2

Will Beta Reduction in different orders get the same result?

Yes: the Church-Rossner Theorem proves that reductions can be done in any order and will always reach the same normal form. Normal forms are unique under full beta reduction. *Caveats abound including "up to alpha conversion", "full beta reduction", and "termination" of reduction.*

Totally Legit Questions 2/2

If the reduction order doesn't matter, why bother talking about different evaluation strategies like Lazy vs Eager evaluation?

1. There are practical effects of eval strategies which will show up in the Lambda Calculus associated with recursion/iteration
2. Real programming languages employ one or the other evaluation strategy or sometimes mixtures of them
   - The programming language Haskell employs a form of Lazy Evaluation (Call by Name) where args to functions only evaluated if they are applied
   - Virtually every other PL (Python, OCaml, C, etc.) employs Eager Evaluation (Call by Value) whee args to functions are evaluated before passing them to functions

Lambda Calculus Programming

- Recall real computers (and Turing machine) use encodings to represent objects of interest like

  `0b0100_1100 = 0x4C = 76 = 'L'`
- To illustrate computational power, must be able to encode a minimal set of objects and functionality, usually
  - Natural numbers (0,1,2,. . .) and addition
  - Booleans and conditional execution
  - Iteration or Recursion
- This is relatively familiar for Real/Turing Machines
- Church showed encodings for each of these in the Lambda Calculus which nets it equal power to Turing Machines
- Variety of other items that can be encoded in Lambda Calculus such as Pairs, Let Bindings, subtraction, multiplication, etc. but the above are enough to show its computational power

Encoding Booleans

- Need two distinct objects which can be used in the same context
- Recall due to Alpha Conversion $\lambda x.x \equiv \lambda y.y$ so they are not distinct
- But these two terms are distinct and can be used in the same contexts (e.g. abstractions that can be applied twice)

$$\lambda x.\lambda y.x : \text{True}$$
$$\lambda x.\lambda y.y : \text{False}$$

- Boolean values ARE if/then/else expressions, provide conditional execution

$$\begin{aligned} &\text{if true then a else b} \\ \Rightarrow &\text{True}\, a\, b \\ \Rightarrow &(\lambda x.\lambda y.x)\, a\, b \\ \Rightarrow &a \end{aligned} \qquad \begin{aligned} &\text{if false then a else b} \\ \Rightarrow &\text{False}\, a\, b \\ \Rightarrow &(\lambda x.\lambda y.y)\, a\, b \\ \Rightarrow &b \end{aligned}$$

Encoding Natural Numbers: Church Numerals

- Again, the informal "type" of the lambda term of all numbers must be the same but each must be distinct
- Achieved via 2 Abstractions with repeated application
- Number of applications corresponds to numeric value

$$\begin{aligned}
0 =& \lambda f.\lambda x.x \\
1 =& \lambda f.\lambda x.f\,x \\
2 =& \lambda f.\lambda x.f\,(f\,x) \\
3 =& \lambda f.\lambda x.f\,(f\,(f\,x)) \\
n =& \lambda f.\lambda x.\text{<applx f n times to x>} \\
n+1 =& \lambda f.\lambda x.f\,(n\,f\,x)
\end{aligned}$$

The IsZero Function

IsZero function: $\lambda z.((z\,(\lambda y.\text{False}))\ \text{True})$

- No reason to expect you could pull this definition out of thin air: that would be miraculous
- BUT demonstrates that a true/false check for a specific numeric value is possible via Beta Reduction

IsZero 0

$\Rightarrow(\lambda \underline{z}.((z\,(\lambda y.\text{False}))\ \text{True}))\ \underline{(\lambda f.\lambda x.x)}$

$\Rightarrow(((\lambda \underline{f}.\lambda x.x)\ \underline{(\lambda y.\text{False})})\ \text{True})$

$\Rightarrow(\lambda \underline{x}.x)\ \underline{\text{True}}$

$\Rightarrow\text{True}$

IsZero 2

$\Rightarrow(\lambda \underline{z}.((z\,(\lambda y.\text{False}))\ \text{True}))\ \underline{(\lambda f.\lambda x.f\ (f\ x))}$

$\Rightarrow((\lambda \underline{f}.\lambda x.f\ (f\ x))\ \underline{(\lambda y.\text{False})})\ \text{True}$

$\Rightarrow(\lambda \underline{x}.(\lambda y.\text{False})\ ((\lambda y.\text{False})\ x))\ \underline{\text{True}}$

$\Rightarrow(\lambda \underline{y}.\text{False})\ \underline{((\lambda y.\text{False})\ \text{True})}$

$\Rightarrow\text{False}$

Encoding Addition

$$\text{Add M N} : \lambda f.\lambda x.(\mathsf{M}\, f\, (\mathsf{N}\, f\, x))$$

- Intent: use Add on two Church numerals M and N
- Note the informal type of Add: 2 abstractions

$$\begin{aligned}
&\text{Add 1 2}\\
\Rightarrow&\lambda f.\lambda x.(\underline{1}\, f\, (2\, f\, x))\\
\Rightarrow&\lambda f.\lambda x.((\lambda \underline{g}.\lambda y.g\, y)\, \underline{f}\, (2\, f\, x))\\
\Rightarrow&\lambda f.\lambda x.(\lambda \underline{y}.f\, y)\, \underline{(2\, f\, x)}\\
\Rightarrow&\lambda f.\lambda x.(f\, (\underline{2}\, f\, x))\\
\Rightarrow&\lambda f.\lambda x.(f\, ((\lambda \underline{g}.\lambda y.g\, (g\, y))\, \underline{f}\, x))\\
\Rightarrow&\lambda f.\lambda x.(f\, ((\lambda \underline{y}.f\, (f\, y))\, \underline{x}))\\
\Rightarrow&\lambda f.\lambda x.(f\, (f\, (f\, x)))\\
\equiv&\ 3
\end{aligned}$$

Church numeral 3 is *double abstraction, apply first arg 3 times to second*

Exercise: Omega Combinator

Consider the following Combinator (closed lambda term)

Omega Combinator: $(\lambda x.x\,x)(\lambda x.x\,x)$

Try reducing this term. . .

Answer: Omega Combinator

Consider the following Combinator (closed lambda term)

Omega Combinator: $(\lambda x.x\,x)(\lambda x.x\,x)$

Try reducing this term...

$$\begin{aligned}
&(\lambda x.x\,x)\,(\lambda x.x\,x)\\
\Rightarrow&(\lambda \underline{y}.y\,y)\,\underline{(\lambda x.x\,x)}\\
\Rightarrow&(\lambda \underline{x}.x\,x)\,\underline{(\lambda x.x\,x)}\\
\Rightarrow&(\lambda \underline{x}.x\,x)\,\underline{(\lambda x.x\,x)}\\
\Rightarrow&(\lambda \underline{x}.x\,x)\,\underline{(\lambda x.x\,x)}\\
&\ldots
\end{aligned}$$

and you'll find the Lambda Calculus has infinite loops built in.

Perhaps it has terminating loops/recursion as well?

Fixed Point Combinator (Y-Combinator)

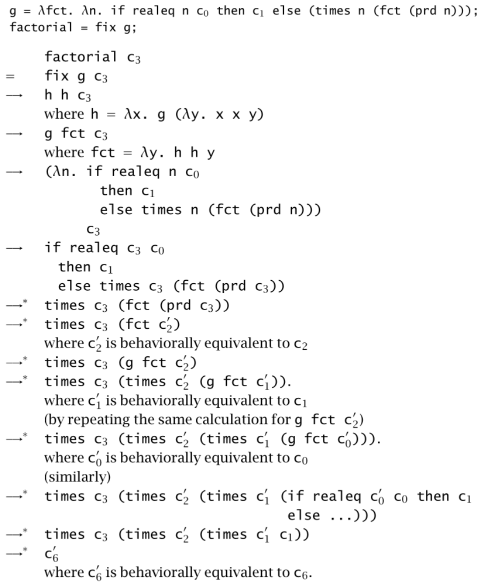
Fix: $\lambda f.(\lambda x.f\,(\lambda y.x\,x\,y))\,(\lambda x.f\,(\lambda y.x\,x\,y))$

Like Omega, the Fix combinator has an intricate, repetitive structure; it is difficult to understand just by reading its definition. Probably the best way of getting some intuition about its behavior is to watch how it works on a specific example
– "Types and Programming Languages" by Pierce

- We won't dwell to much on the intricacies of Fix
- TL;DR version: it allows the parameter f to replicate within the combinator, then replicate again, then again...
- Some folks refer to this as "recursion"
- A more literal interpretation is just substituting the body again
- End game: Fixed Point Combinator makes Lambda Calculus Turing Complete

Fixed Point Reductions

- Assume you can define multiplication and decrement (subtract 1) combinators for Church Numerals in Lambda Calc
- With these and the Fixed Point Combinator in hand, can define a **Factorial Function** (goto example in *every* Lambda Calc tutorial)
- Calculations are tedious but possible



g = λfct. λn. if realeq n c_0 then c_1 else (times n (fct (prd n)));
factorial = fix g;

factorial c_3
= fix g c_3
$\longrightarrow$ h h c_3
where h = λx. g (λy. x x y)
$\longrightarrow$ g fct c_3
where fct = λy. h h y
$\longrightarrow$ (λn. if realeq n c_0
then c_1
else times n (fct (prd n)))
c_3
$\longrightarrow$ if realeq c_3 c_0
then c_1
else times c_3 (fct (prd c_3))
$\longrightarrow^*$ times c_3 (fct (prd c_3))
$\longrightarrow^*$ times c_3 (fct c_2')
where c_2' is behaviorally equivalent to c_2
$\longrightarrow^*$ times c_3 (g fct c_2')
$\longrightarrow^*$ times c_3 (times c_2' (g fct c_1')).
where c_1' is behaviorally equivalent to c_1
(by repeating the same calculation for g fct c_2')
$\longrightarrow^*$ times c_3 (times c_2' (times c_1' (g fct c_0'))).
where c_0' is behaviorally equivalent to c_0
(similarly)
$\longrightarrow^*$ times c_3 (times c_2' (times c_1' (if realeq c_0' c_0 then c_1
else ...)))
$\longrightarrow^*$ times c_3 (times c_2' (times c_1' c_1))
$\longrightarrow^*$ c_6'
where c_6' is behaviorally equivalent to c_6.

Figure 5-2: Evaluation of factorial c_3

Source: Pierce Ch 5

Issues with Substitutions

Name Capture

Consider the following naive reduction:

$$(\lambda \underline{x}.\lambda y.x\,y)\,\underline{y}$$
$$\Rightarrow \lambda y.y\,y$$

Problematic as the y vars are different

- Left y is a bound variable
- Right y is a free variable

Due to unfortunate naming, free y is **captured** by the parameter y
Remedied via Alpha-Conversion: rename away from free variables

$$(\lambda x.\lambda \underline{y}.x\,\underline{y})\,y$$
$$\equiv(\lambda \underline{x}.\lambda z.x\,z)\,\underline{y}$$
$$\Rightarrow \lambda z.y\,z$$

Substitutions / Environments

When coding Lambda Calculus interpreters, substitutions are often facilitated with **environments**: a list of variable / value pairs

$$(\lambda x.t_1)\,t_2 \Rightarrow [x \mapsto t_2]t_1$$

Substitutions stack:

$$(\lambda y.(\lambda x.t_1)\,t_2)\,t_3 \Rightarrow [y \mapsto t_3, x \mapsto t_2]\,t_1$$

Environments often coded as a data structures

- Association Lists in simple implementations

```
let env = [(name1,val1);
           (name2,val2); ...]
```

- Hash tables in more sophisticated code that needs speed for lookup

A Note on Definitions and the Difficulties they Cause

Post 1376: *In Prof. Kauffman's slide (Lambda Calculus), p18, when we do exercise C with an Eager Evaluation in Lambda Calculus. Why would the answer be calculated that way?*

TA: You are correct that the notes are not an Eager Evaluation, and the first step should have been to evaluate the rightmost expression.

```
(Lx.x w) ((Ly.y) (Lz.((Lu.u) z)))
 ^-e1-^  ^----------e2----------^

Eager, evaluate e2 first
(Ly.y) (Lz.((Lu.u) z)
^-e3-^  ^----e4-----^


Eager, evaluate e4 first
```

Kauffman: I disagree. You are using a different definition of Eager Evaluation than what I presented.

- Pierce's "Call by Value": `e4` is a value, does not reduce
  Equivalent to `let foo n = 1+2+n;;` and don't evaluate `1+2` until the function is actually called
- Hicks's "Call by Value": descend into `e4` and reduce its body
  Equivalent to `let foo n = 1+2+n;;` AND perform partial evaluation / compiler optimization to transform it to `let foo n = 3+n;;`

References for Definitions

"Pierce"

Types and Programming Languages, Ch 5 by Benjamin C. Pierce

- Defines Call by Value Evaluation Strategy with Lambda terms as values, no partial evaluation
- Book on programming languages and types developed with input from an array of PL researchers, targeted at grad students studying programming language theory, self-contained text with references, code implementations, etc.
- E.g. a published textbook, an authoritative source

"Hicks"

These slides: `https://www.cs.umd.edu/class/spring2021/cmsc330/lectures/24-lambda-calc-1.pdf`

- A past offering of CMSC330 (`https://www.cs.umd.edu/class/spring2021/cmsc330/`)
- Anwar Mamat and Michael Hicks listed as instructors, both are now at Amazon in web services development
- No author attributed to slides, no references in slides, no peer or editor review; not wrong, just less authoritative

How about on Project 6 and Quizzes / Exams?

To avoid confusion, we will avoid test cases in which partial evaluation would lead to different results.
The behavior of reducing `(Lz.((Lu.u) z)` is unspecified: it may or may not reduce further.