CMSC330: The Lambda Calculus

Chris Kauffman

Last Updated:
Tue Oct 24 09:21:18 AM EDT 2023

Logistics

Assignments

- Project 5 up and running
- NFA to DFA conversion in OCaml
- P5 due 30-Oct

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

Survey of Lambda Calculus

Announcements

None

Church-Turing Thesis

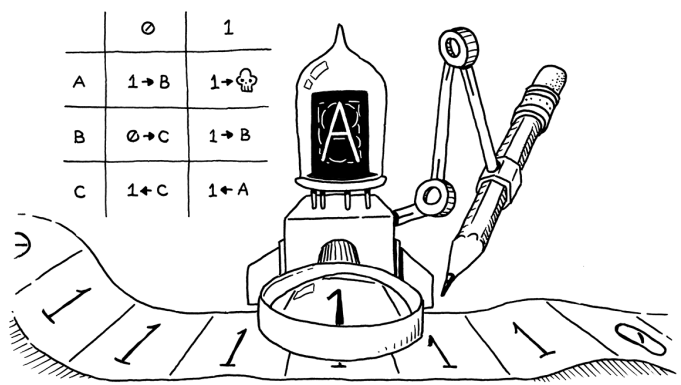
- Mathematicians wished to formalize the notion of an "algorithm"
- A variety of different models were proposed the best know of which are
  - The Lambda Calculus by Alonzo Church
  - Turing Machines by Alan Turing (originally called "a-machines"
- Work by Turing, Church, Stephen Kleene (ring any bells?) 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

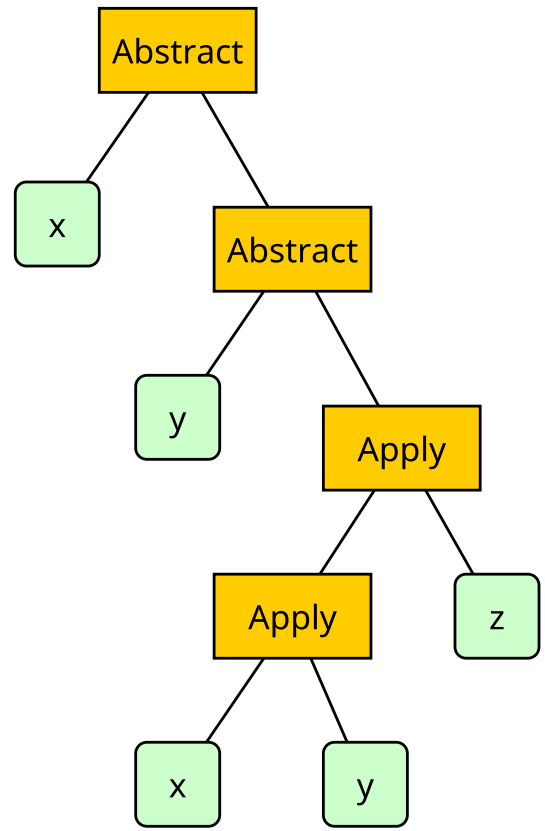
Abstraction body is implicitly parenthesized

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

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 $\lambda x. \ldots x \ldots$
- 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 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

== END PART 1 ==

Eager vs Lazy Evaluation Strategies

Lazy

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

Eager / Strict Evaluation

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

Encoding Booleans

Encoding Pairs

Encoding Numbers: Church Numerals

Encoding Addition

Recursion in the Lambda Calculus

Omega Combinator

Y-Combinator and

Substitutions