15-252: More Great Theoretical Ideas in Computer Science

Lecture 3
January 30, 2020

Lambda Calculus

Alonzo Church

What is a computation/algorithm?

Hilbert's 10th problem (1900):

Given a multivariate polynomial w/ integer coeffs,
e.g. $4x^2y^3 - 2x^4z^5 + x^8$,
"*devise a process according to which it can be determined in a finite number of operations*"
whether it has an integer root.

Mathematicians: "we should probably try to
formalize what counts as
an 'algorithm' ".

What is a computation/algorithm?

Hilbert's Entscheidungsproblem (1928):

("decision problem")

Given a sentence in first-order logic,
give an "*effectively calculable procedure*"
for determining if it's provable.

Mathematicians: "we should probably try to
formalize what counts as
an 'algorithm' ".

Gödel (1934):

Discusses some ideas for definitions of what functions/languages are “computable”, but isn’t confident what’s a good definition.

Church (1936):

Invents lambda calculus, claims it should be the definition.

Gödel, Post (1936):

Arguments that Church’s claim isn’t justified.

Meanwhile… in Princeton… a certain British grad student of Alonzo Church…

Alan Turing (1936, age 22):

Describes a new model of computation, now known as the **Turing Machine**.

Emil Post (1936), independently, proposed an especially simple type of TM (2-way infinite tape, each cell holding a bit), which was later proved to be equivalent to Turing Machines.

Turing's model gained acceptance as a right general mathematical model of computability

1937: Turing proves TMs = lambda calculus

Today we’re going to discuss this alternate model of computing called Lambda Calculus.

It’s a simple programming language invented in the 1930s by Alonzo Church, for the purpose of defining computability.

Why study Lambda Calculus?

Historically it preceded Turing Machines.

It is elegant, and offers an alternate higher level perspective on computation.

It's the simplest known sufficiently powerful programming language.

It has direct relevance to programming; modern functional programming languages (e.g. ML, Scheme, Haskell, F#, Scala) are direct descendants of lambda calculus.

Church–Turing Thesis:

Any natural / reasonable notion of computation realizable in the physical world can be simulated by a TM (or equivalently, by lambda calculus).

Outline

- Syntax of Lambda Calculus
- Calculating with Lambda Calculus
- Uniqueness Theorems & Church-Rosser Property
- How to implement recursion
- Implementing built-in constants and functions

Calculus?

- Calculus = just a bunch of rules for manipulating symbols
- One can give meaning to the symbols (semantics), but that's not part of the calculus (pure syntax)
- Lambda calculus: rules for manipulating expressions of form <exp> ::= <constant> | <variable>| <exp> <exp> | λ <variable> . <exp>| (<exp>)
- One can associate meanings to expressions in a way that corresponds to computations (functional programs)
  - We will motivate the calculus/syntax via the intended semantics

A functional program is an *expression* which is 'executed' by *evaluating* it.

For example: (+ 4 5)

+ is a function. We write functions in prefix form.

Another example: (+ (* 5 6) (* 8 3))

Evaluation proceeds by choosing a reducible expression and reducing it. (There may be more than one order.)

(+ (* 5 6) (* 8 3)) ➞ (+ 30 (* 8 3)) ➞ (+ 30 24) ➞ 54

Function Application and Currying

Function application is indicated by juxtaposition: f x

“The function f applied to the argument x”

What if we want a function of more than one argument? We could invent a notation f(x,y), but there’s an alternative. To express the sum of 3 and 4 we write:

((+ 3) 4)

The expression (+ 3) denotes the function that adds 3 to its argument.

Function Application and Currying (contd.)

So all functions take one argument. So when we wrote (+ 3 4) this is shorthand for ((+ 3) 4). (The arguments are associated to the left.)

This mechanism is known as currying (in honor of Haskell Curry).

Parentheses can be added for clarity. E g:

((f ((+ 4) 3)) (g x)) is identical to f (+ 4 3) (g x)

Built-in Functions and Constants

- We will start out with a version of the lambda calculus that has constants like 0, 1, 2… and functions such as + * =.
- We also include constants TRUE and FALSE, and logical functions AND, OR, NOT.

Also conditional expressions, with these reduction rules:

$$\text{IF TRUE } E_1\ E_2 \rightarrow E_1$$

$$\text{IF FALSE } E_1\ E_2 \rightarrow E_2$$

(Later we'll see that the expressive power of our language is not enhanced by any of these, by showing how to `implement' these.)

Writing Your Own Functions – Lambda Abstractions

A way to write expressions that denote functions.

Example: (λx . + x 1)

λ means here comes a function

Then comes the parameter x

Then comes the .

Then comes the body + x 1

The function is ended by the) (or end of string)

(λx . + x 1) is the increment function.

$$(\lambda x\ .\ +\ x\ 1)\ 5 \rightarrow 6$$

(We'll explain this more thoroughly later.)

Syntax Summary

<exp> ::= <constant>	Built-in constants & functions
| <variable>	Variable names, e.g. x, y, z…
| <exp> <exp>	Application
| λ <variable> . <exp>	Lambda Abstractions
| (<exp>)	Parens

Example: λf.λn. IF (= n 0) 1 (* n (f (- n 1)))

Implied paranthesis in lambda expressions

Functions associate to the left: f g x stands for (f g) x

Lambdas associate to the right:
λx. x 3 stands for λx. (x 3),
i.e., function which takes x and applies it to 3

Note (λx. x) 3 equals 3 (identity function applied to 3)

Eg: λx.λy. x y (λz. z y) when fully paranthesized is

λx.λy. ((x y) (λz. (z y)))

Sometimes abbreviate λx.λy by λxy.

Bound and unbound variables

Consider this lambda expression:

$(\lambda x.\ +\ x\ y)\ 4$

x is a *bound variable*, because it's inside the body of the λx expression.

y is an *unbound or free variable*, because it's not in a λy expression.

In this expression $+\ x\ ((\lambda x.\ +\ x\ 1)\ 4)$ The first x is free, and the second one is bound. In $(+\ x\ 1)$, the x is free.

β-Reduction (aka λ expression evaluation)

Consider this lambda expression:

$$(\lambda x.\ +\ x\ 1)\ 4$$

This juxtaposition of $(\lambda x.\ +\ x\ 1)$ with 4 means to apply the lambda abstraction $(\lambda x.\ +\ x\ 1)$ to the argument 4. Here's how we do it:

The result of applying a lambda abstraction to an argument is an instance of the body of the lambda abstraction in which bound occurrences of the formal parameter in the body are replaced with copies of the argument.

I.e. we put the 4 in for the x in $+\ x\ 1$. The result is $+\ 4\ 1$.

$$(\lambda x.\ +\ x\ 1)\ 4 \rightarrow_{\beta} +\ 4\ 1$$

β-Reduction (some more examples)

$(\lambda x.\ +\ x\ x)\ 5 \rightarrow +\ 5\ 5 \rightarrow 10$

$(\lambda x.\ 3)\ 5 \rightarrow 3$

$(\lambda x.\ (\lambda y.\ -\ y\ x))\ 4\ 5 \rightarrow (\lambda y.\ -\ y\ 4)\ 5 \rightarrow -\ 5\ 4 \rightarrow 1$

(Note the currying – we peel off the arguments 4 then 5.)

$(\lambda f.\ f\ 3)\ (\lambda x.\ +\ x\ 1) \rightarrow (\lambda x.\ +\ x\ 1)\ 3 \rightarrow +\ 3\ 1 \rightarrow 4$

$$\begin{aligned}(\lambda x.\ (\lambda x.\ +\ (-\ x\ 1))\ \ x\ 3)\ 9 &\rightarrow (\lambda x.\ +\ (-\ x\ 1))\ 9\ 3\\ &\rightarrow +\ (\ -\ 9\ 1)\ 3\\ &\rightarrow 11\end{aligned}$$

β-Reduction (contd)

$(\lambda x.\ (\lambda x.\ + (- x\ 1))\ \ x\ 3)\ 9 \rightarrow (\lambda x.\ + (- x\ 1))\ 9\ 3$

$\rightarrow + (\ - 9\ 1)\ 3$

$\rightarrow 11$

In this last example we could have applied the reduction in a different order.

$(\lambda x.\ \underline{(\lambda x.\ + (- x\ 1))\ \ x\ }\ 3)\ 9 \rightarrow (\lambda x.\ + (- x\ 1)\ 3\ )\ 9$

$\rightarrow + (\ - 9\ 1)\ 3$

$\rightarrow 11$

The other kinds of conversions

There are some other (less interesting) kinds of reductions that we shall not spend much time on.

α-conversion: $(\lambda x.\ +\ x\ x) \rightarrow_{\alpha} (\lambda y.\ +\ y\ y)$

(just renaming locally bound variables to avoid confusion)

δ-reductions: applying the built in functions

η-reductions: $\lambda f.\ \lambda x\ f\ x \rightarrow_{\eta} \lambda f.\ f$

β-Reduction (ordering)

So, to evaluate an expression we search for reductions, and make them. When the process stops (there are no more reductions to apply), the expression is said to be in ***normal form***. This is the *value* of the original expression.

Wait… there are different orders in which to do the reductions. Does the order matter? If so, we have a problem.

β-Reduction (ordering)

Let D be this expression: (λx. x x). What is the value of this expression (D D)?

(D D) = (λx. x x) (λx. x x) ➞ (λx. x x) (λx. x x) ➞ …

It never terminates!

Now consider this expression: (λx. 3) (D D)

If we apply β-reduction to the leftmost part first, we get 3. If we keep applying it to (D D) we go into an infinite loop.

Former is *call-by-name* (lazy evaluation)
Latter is *call-by-value* (eager evaluation)

On the ordering of reductions

Church-Rosser Theorem : No expression can be converted into two distinct normal forms.

(Modulo renaming .. I.e (λx. + x x) and (λy. + y y) the same.)

Note: evaluations for some ordering may not terminate,
but will never terminate leading to a different normal form.

Normal Order: do the *leftmost outermost* reduction first.

Church-Rosser Theorem 2: if $E_1 \rightarrow^* E_2$ and E_2 is in normal form, then you can get to E_2 by reducing in normal order.

So if we just use normal order, we're good.
We will stick with normal order from now on.

The Church-Rosser Property (sidebar)

Consider the following problem. You're given an undirected graph G. You want to find a subgraph of G such that all vertices in it have degree at least 4 (with respect to the given subgraph).

Can you think of an algorithm to find the maximum such subgraph?

Solution: Greed: Find any vertex with current degree < 4. Delete it. Repeat.

<u>Exercise theorem</u>: Any run of this algorithm will find the same solution, which is the unique optimal one.

The Church-Rosser Property

This is an example of a *reduction system* with the *Church-Rosser Property*.

Roughly speaking, a reduction system is where you have state and rules to change the state. It stops when there are no more rules that apply.

A reduction system is Church-Rosser if no matter how you apply the rules, *if it terminates*, then you end up in the same state.

This is the situation with lambda calculus reductions, as Church and Rosser proved.

Implementing recursion in λ calculus

We claimed that lambda calculus was powerful.
We've seen how to define expressions.
But the language does not seem to support loops or recursive calls.

All functions are anonymous. There is no mechanism for naming a function, then calling it by its name.

But one can get around this problem.

Implementing recursion

Suppose we wanted to write a factorial function which takes a number n and computes n!   Here's an attempt to get started:

$$\lambda n.\text{if } (= n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))$$

This does not work. Because what is the unbound variable f ? It would work if we could somehow make f be the very function above.

Implementing recursion

Idea: To give us access to that function, how about passing it in as another parameter?

$(\lambda f.\lambda n.\text{if } (= n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))$

- But we need to pass this function to itself (recursively).
- So inner recursive call should be (f f (- n 1)).

Implementing recursion

To give us access to a function itself, we pass it in as another parameter.

FACT = (λf.λn.if (= n 0) 1 (* n (f f (- n 1))))

(FACT is just shorthand for that string of characters.)

Now if we write

FACT FACT 5

This will work, because the β-reduction substitutes FACT for f, resulting in a function call FACT FACT 4. Etc.

Implementing recursion

Another example. The Fibonacci numbers

FIB = λf.λn.if (= n 0) 0 (if (= n 1) 1 (+ (f f (- n 1)) (f f (- n 2))))

Now FIB FIB n computes the nth Fibonacci number.

A general method for recursion: the Y combinator

There's a systematic way to do this. We'll just touch upon this, but you can see http://en.wikipedia.org/wiki/Fixed-point_combinator

Let H be the factorial function we attempted to use before:

$$H = \lambda f.\lambda n.\text{if } (= n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))$$

Let Y be the following expression.

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

This is called a *fixedpoint combinator* because for any G:

$$Y\ G = G\ (Y\ G)$$

Turns out Y H is the factorial function that we wanted.

Exercise: ***Check this!***

Building in constants and functions

We wany to represent things, like for example, numbers and Booleans.

All these can be defined using λ-abstractions.

We will see some examples next.

Examples of Simple Normal-Form Expressions

$\lambda x.\lambda y.x$ $\qquad\qquad$ $\lambda x.\lambda y.y$

Let's look at the first one. What does it do when applied?

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

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

If we call them TRUE and FALSE respectively, then we have replaced our IF construct.

IF FOO B_1 B_2 $\qquad$ replaced by $\qquad$ FOO B_1 B_2

TRUE = λx.λy.x FALSE = λx.λy.y

We'll need the boolean operations, like AND, OR, NOT. How would we do AND?

Essentially we use our IF-THEN-ELSE design.

AND A B = IF A then B else FALSE

AND A B = A B FALSE

AND = λa.λb. a b FALSE = λa.λb. a b (λx.λy.y)

Actually we can also write AND more simply as

AND = λa.λb. a b a

Easy exercise: Work out OR and NOT.

What about Numbers?

Look at these normal form expressions….

λf.λx x	0
λf.λx f x	1
λf.λx f (f x)	2
λf.λx f (f (f x))	3
λf.λx f (f (f (f x)))	4

These are called the Church numerals

n f x applies function f iteratively n times to x.

To be useful as numbers we have to be able to test for zero, add, subtract, multiply, etc

Testing a number for zero

λf.λx x	0
λf.λx f x	1
λf.λx f (f x)	2
λf.λx f (f (f x))	3
λf.λx f (f (f (f x)))	4

Let g = λx. FALSE.

Then for any numeral n

n g TRUE = TRUE if and only if n=0

Therefore we have the function

ISZERO = λn. n (λx. FALSE) TRUE

Adding Numbers

If n and m are numbers, then “m+n” f x applies the function f m+n times to x. How do we write such a function?

Easy. It’s m f (n f x), so just put that into lambda form:

$$\text{SUM} = \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)$$

Multiplying Numbers

What we're going to do here is replace the function used by m with one which repeats f n times. So we end up with f being applied m*n times.

(n f) is a function that takes one parameter and applies f n times to it.

So (m (n f)) is a function that takes one parameter and applies f m*n times to it.

PROD m n = λf.λx. m (n f) x

PROD = λm.λn.λf.λx. m (n f) x

Subtraction

To compute m-n, apply the predecessor function n times to m.

SUBTRACT := $\lambda m.\ \lambda n.\ n$ PRED m

How to compute predecessor?

- This is not so straightforward.
- Can implement using "pairs"

Data Structures - Pairs

Let E, F be arbitrary normal form expressions.
A pair (E,F) can be represented as the normal form lambda expression: $\lambda f.\ f\ E\ F$

The function to construct a pair is then:

$$\text{PAIR} := \lambda x.\lambda y.\lambda f.\ f\ x\ y$$

If p is a pair (E,F), i.e., $\lambda f.\ f\ E\ F$ then how do we get its parts?

Easy. p TRUE gives us E, and p FALSE gives us F.

FIRST := λp.p TRUE SECOND := λp.p FALSE

Data Structures - Lists

Can turn our method of pairing arbitrary normal form expressions into building up lists.

All we need is a way to mark the end of the list (NIL) and a way to tell if we're at the end of the list (ISEMPTY).

NIL := λx.TRUE   ISEMPTY := $\lambda p.p$ ($\lambda x.\lambda y$.FALSE)

Let's see how this works:

ISEMPTY NIL = ($\lambda p.p$ ($\lambda x.\lambda y$.FALSE)) (λx.TRUE)
$\rightarrow$ (λx.TRUE) ($\lambda x.\lambda y$.FALSE) $\rightarrow$ TRUE

ISEMPTY (PAIR E F) = ($\lambda p.p$ ($\lambda x.\lambda y$.FALSE)) (λf. f E F)
$\rightarrow$ (λf. f E F) ($\lambda x.\lambda y$.FALSE)
$\rightarrow$ ($\lambda x.\lambda y$.FALSE) E F $\rightarrow$ FALSE

Data Structures – Lists (contd)

So, for example, we can build the list [0,1,2] as follows:

PAIR 0 (PAIR 1 (PAIR 2 NIL))

Expanding our shorthand into pure lambda calculus, this is:

λf.f (λf.λx.x) (λf.f (λf.λx. f x) (λf.f (λf.λx.f (f x)) (λx.λz.λy.z)))

One could continue to develop more data structures. But we'll stop here.

Predecessor

To compute m-n, apply the predecessor function n times to m.

SUBTRACT := λ*m*.λ*n*.*n* PRED *m*

Predecessor using pairs:

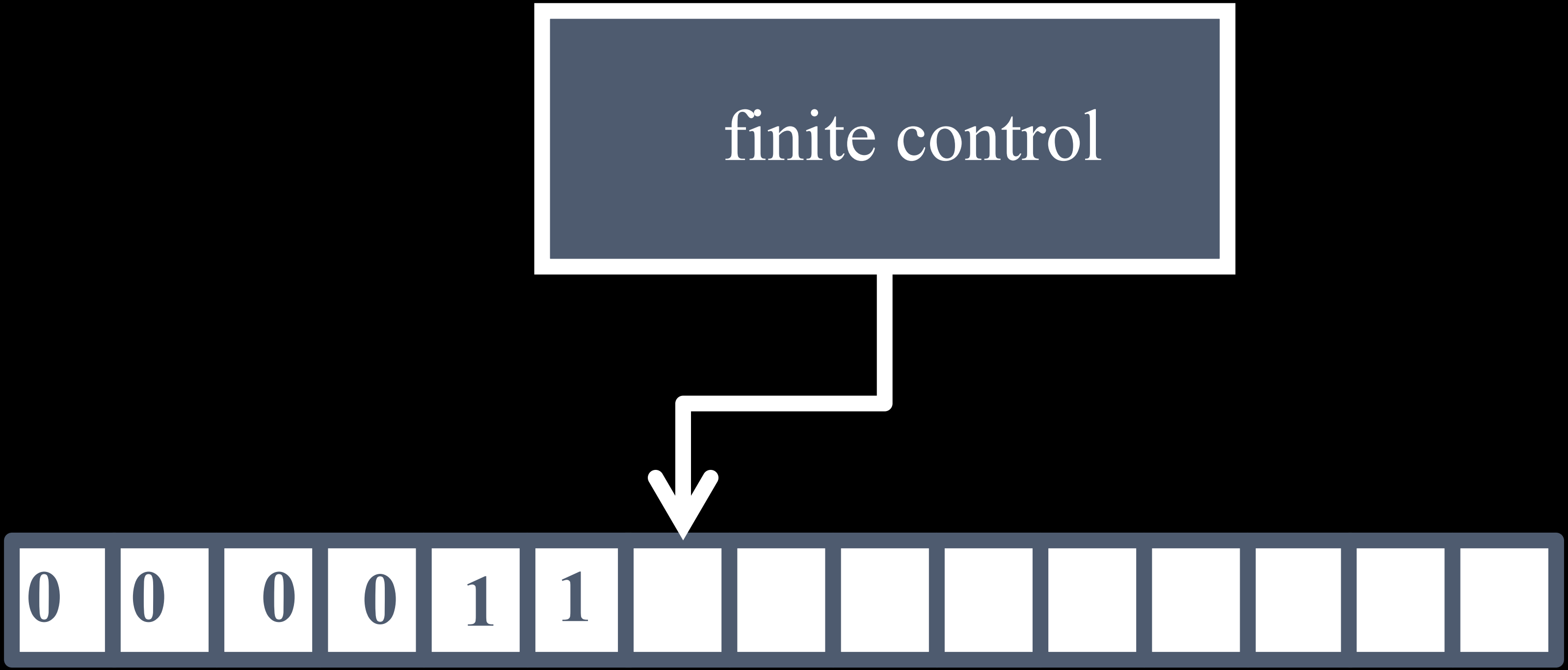
PRED := λn. FIRST (n NEXT (PAIR 0 0))

NEXT := λp. PAIR (SECOND p) (SUM 1 (SECOND p))

Lambda Calculus is cool!

Having built all the basic primitives one can prove (though we won't do it here):

Theorem : The computing power of lambda calculus is the same as the Turing machine.



Implementing a TM in Lambda Calculus:

- Read/write infinite tape: Lists
- Finite control: Numbers to keep track of state
- Processing:
  - Way to make decisions (if)
  - Way to keep going (loop)

Recap

- Church-Turing thesis
- Syntax of Lambda Calculus
- Evaluating expressions
- Uniqueness Theorems
- How to implement recursion
- Booleans
- Numbers
- Doing Arithmetic
- Pairs & Lists