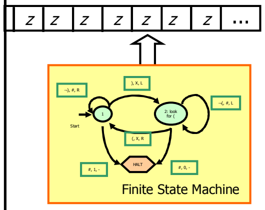
Lecture 39: Lambda Calculus

CS150: Computer Science
University of Virginia
Computer Science

David Evans
http://www.cs.virginia.edu/evans

Equivalent Computers



Turing Machine

≡

$$\begin{aligned} term = \ & variable \\ & \mid term\ term \\ & \mid (term) \\ & \mid \boldsymbol{\lambda}\, variable\ .\ term \end{aligned}$$

$\lambda y.\ M \Rightarrow_{\alpha} \lambda v.\ (M\ [y\ \alpha\ \ v])$
where v does not occur in M.

$(\lambda x.\ M)N \Rightarrow_{\beta} M\ [\ x\ \alpha\ N\ ]$

Lambda Calculus

What is Calculus?

- In High School:

$d/dx\ x^n = nx^{n-1}$ [Power Rule]

$d/dx\ (f + g) = d/dx\ f + d/dx\ g$ [Sum Rule]

Calculus is a branch of mathematics that deals with limits and the differentiation and integration of functions of one or more variables...

Real Definition

- A *calculus* is just a bunch of rules for manipulating symbols.
- People can give meaning to those symbols, but that's not part of the calculus.
- Differential calculus is a bunch of rules for manipulating symbols. There is an interpretation of those symbols corresponds with physics, slopes, etc.

Lambda Calculus

- Rules for manipulating strings of symbols in the language:

$$\begin{aligned} term = \ & variable \\ & \mid term\ term \\ & \mid (term) \\ & \mid \boldsymbol{\lambda}\, variable\ .\ term \end{aligned}$$

- Humans can give meaning to those symbols in a way that corresponds to computations.

Why?

- Once we have precise and formal rules for manipulating symbols, we can use it to reason with.
- Since we can interpret the symbols as representing computations, we can use it to reason about programs.

Evaluation Rules

α-reduction (renaming)

$\lambda y.\ M \Rightarrow_{\alpha} \lambda v.\ (M\ [y\ \alpha\ \ v])$

where v does not occur in M.

β-reduction (substitution)

$(\lambda x.\ M)N \Rightarrow_{\beta} M\ [\ x\ \alpha\ \ N\ ]$

Note the syntax is different from Scheme:
$(\lambda x.M)N \rightarrow$ ((lambda (x) M) N)

β-Reduction (the source of all computation)

$$(\lambda x.\ M)N \rightarrow M\ [\ x\ \alpha\ \ N\ ]$$

Replace all x's in M witn N

Evaluating Lambda Expressions

- *redex*: Term of the form $(\lambda x.\ M)N$

  Something that can be β-reduced
- An expression is in *normal form* if it contains no redexes (*redices*).
- To evaluate a lambda expression, keep doing reductions until you get to *normal form*.

Some Simple Functions

$\mathbf{I} \equiv \lambda x.x$

$\mathbf{C} \equiv \lambda xy.yx$

Abbreviation for $\lambda x.(\lambda y.\ yx)$

$\mathbf{CII} = (\lambda x.(\lambda y.\ yx))\ (\lambda x.x)\ (\lambda x.x)$

$\rightarrow_{\beta} (\lambda y.\ y\ (\lambda x.x))\ (\lambda x.x)$

Example

$$\lambda\ f.\ ((\lambda\ x.f\ (xx))\ (\lambda\ x.\ f\ (xx)))$$

Possible Answer

$(\lambda\ f.\ ((\lambda\ x.f\ (xx))\ (\lambda\ x.\ f\ (xx))))\ (\lambda z.z)$

$\rightarrow_{\beta}\ (\lambda x.(\lambda z.z)(xx))\ (\lambda\ x.\ (\lambda z.z)(xx))$

$\rightarrow_{\beta} (\lambda z.z)\ (\lambda\ x.(\lambda z.z)(xx))\ (\lambda\ x.(\lambda z.z)(xx))$

$\rightarrow_{\beta} (\lambda x.(\lambda z.z)(xx))\ (\lambda\ x.(\lambda z.z)(xx))$

$\rightarrow_{\beta} (\lambda z.z)\ (\lambda\ x.(\lambda z.z)(xx))\ (\lambda\ x.(\lambda z.z)(xx))$

$\rightarrow_{\beta} (\lambda x.(\lambda z.z)(xx))\ (\lambda\ x.(\lambda z.z)(xx))$

$\rightarrow_{\beta} \ldots$

Alternate Answer

$$(\lambda\, f.\ ((\lambda\ x.f\,(xx))\ (\lambda\ x.\ f\,(xx))))\ (\lambda z.z)$$
$$\rightarrow_\beta (\lambda x.(\lambda z.z)(xx))\ (\lambda\ x.\ (\lambda z.z)(xx))$$
$$\rightarrow_\beta (\lambda x.xx)\ (\lambda x.(\lambda z.z)(xx))$$
$$\rightarrow_\beta (\lambda x.xx)\ (\lambda x.xx)$$
$$\rightarrow_\beta (\lambda x.xx)\ (\lambda x.xx)$$
$$\rightarrow_\beta \ldots$$

Be Very Afraid!

- Some λ-calculus terms can be β-reduced forever!
- The order in which you choose to do the reductions might change the result!

Take on Faith (until CS655)

- All ways of choosing reductions that reduce a lambda expression to normal form will produce the same normal form (but some might never produce a normal form).
- If we always *apply the outermost lambda first*, we will find the normal form if there is one.
  - This is *normal order reduction* – corresponds to normal order (lazy) evaluation

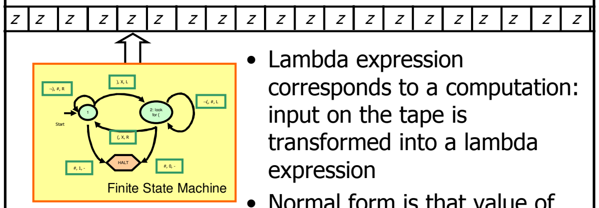
Universal Language

- Is Lambda Calculus a *universal language*?
  - Can we compute any computable algorithm using Lambda Calculus?
- To prove it isn't:
  - Find some Turing Machine that cannot be simulated with Lambda Calculus
- To prove it is:
  - Show you can simulate *every* Turing Machine using Lambda Calculus

Simulating Every Turing Machine

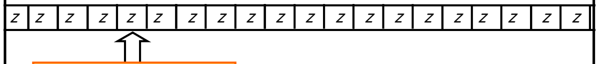
- A Universal Turing Machine can simulate every Turing Machine
- So, to show Lambda Calculus can simulate every Turing Machine, all we need to do is show it can simulate a Universal Turing Machine

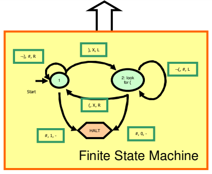
Simulating Computation



- Lambda expression corresponds to a computation: input on the tape is transformed into a lambda expression
- Normal form is that value of that computation: output is the normal form
- How do we simulate the FSM?

Simulating Computation





Finite State Machine

Read/Write Infinite Tape
    Mutable Lists
Finite State Machine
    Numbers
Processing
    Way to make decisions (if)
    Way to keep going

Making "Primitives" from Only Glue (λ)

In search of *the truth*?

- What does **true** mean?
- **True** is something that when used as the first operand of **if**, makes the value of the **if** the value of its second operand:

  if T $M\ N \rightarrow M$

Don't search for **T**, search for **if**

$$\mathbf{T} \equiv \lambda x\ (\lambda y.\ x)$$
$$\equiv \lambda xy.\ x$$
$$\mathbf{F} \equiv \lambda x\ (\lambda\ y.\ y))$$
$$\mathbf{if} \equiv \lambda pca\ .\ pca$$

Finding the Truth

$\mathbf{T} \equiv \lambda x\ .\ (\lambda y.\ x)$

$\mathbf{F} \equiv \lambda x\ .\ (\lambda y.\ y)$

$\mathbf{if} \equiv \lambda p\ .\ (\lambda c\ .\ (\lambda a\ .\ pca)))$ Is the **if** necessary?

$\mathbf{if}\ \mathbf{T}\ \boldsymbol{M}\ \boldsymbol{N}$

$((\lambda pca\ .\ pca)\ (\lambda xy.\ x))\ \boldsymbol{M}\ \boldsymbol{N}$

$\rightarrow_\beta (\lambda ca\ .\ (\lambda x.(\lambda y.\ x))\ ca))\ \boldsymbol{M}\ \boldsymbol{N}$

$\rightarrow_\beta \rightarrow_\beta (\lambda x.(\lambda y.\ x))\ \boldsymbol{M}\ \boldsymbol{N}$

$\rightarrow_\beta (\lambda y.\ \boldsymbol{M}\ ))\ \boldsymbol{N} \rightarrow_\beta \boldsymbol{M}$

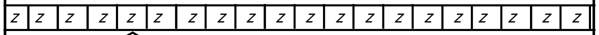
---

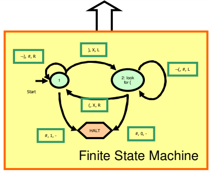
and and **or**?

$\mathbf{and} \equiv \lambda x\ (\lambda y.\ \mathbf{if}\ x\ y\ \mathbf{F}))$

$\mathbf{or} \equiv \lambda x\ (\lambda y.\ \mathbf{if}\ x\ \mathbf{T}\ y))$

Lambda Calculus is a Universal Computer?





- Read/Write Infinite Tape
  - **Mutable Lists**
- Finite State Machine
  - **Numbers**
- Processing
  - **Way to make decisions (if)**
  - **Way to keep going**

What is 42?

42

forty-two

XLII

cuarenta y dos

Meaning of Numbers

- "42-ness" is something who's **successor** is "43-ness"
- "42-ness" is something who's **predecessor** is "41-ness"
- "Zero" is special. It has a **successor** "one-ness", but no **predecessor**.

Meaning of Numbers

pred (succ N) $\rightarrow$ N
succ (pred N) $\rightarrow$ N
succ (pred (succ N)) $\rightarrow$ succ N

zero? **zero** $\rightarrow$ **T**
zero? (succ **zero**) $\rightarrow$ **F**

Is this enough?

Can we define **add** with **pred**, **succ**, **zero?** and **zero**?

$$\textbf{add} \equiv \lambda xy.\textbf{if}\ (\textbf{zero?}\ x)\ y\ (\textbf{add}\ (\textbf{pred}\ x)\ (\textbf{succ}\ y))$$

Can we define lambda terms that behave like **zero**, **zero?**, **pred** and **succ**?

Hint: what if we had **cons**, **car** and **cdr**?

Numbers are Lists...

zero? ≡ **null?**

pred ≡ **cdr**

succ ≡ $\lambda\ x$. **cons F** x

The *length* of the list corresponds to the number value.

Making Pairs

```
(define (make-pair x y)
  (lambda (selector) (if selector x y)))

(define (car-of-pair p) (p #t))
(define (cdr-of-pair p) (p #f))
```

cons and car

cons ≡ $\lambda x.\lambda y.\lambda z.zxy$

cons M N = $(\lambda x.\lambda y.\lambda z.zxy)$ M N

$\rightarrow_\beta (\lambda y.\lambda z.z\mathrm{M}y)\ \mathrm{N}$

$\rightarrow_\beta \lambda z.z\mathrm{MN}$

car ≡ $\lambda p.p\ \mathbf{T}$

$\mathbf{T} \equiv \lambda xy.\ x$

car (cons M N) ≡ car $(\lambda z.z\mathrm{MN}) \equiv (\lambda p.p\ \mathbf{T})\ (\lambda z.z\mathrm{MN})$

$\rightarrow_\beta (\lambda z.z\mathrm{MN})\ \mathbf{T} \rightarrow_\beta \mathbf{T}\mathrm{MN}$

$\rightarrow_\beta (\lambda xy.\ x)\ \mathrm{MN}$

$\rightarrow_\beta (\lambda y.\ \mathrm{M})\mathrm{N}$

$\rightarrow_\beta \mathrm{M}$

cdr too!

cons ≡ $\lambda xyz.zxy$

car ≡ $\lambda p.p$ T

cdr ≡ $\lambda p.p$ F

cdr cons $M\ N$

cdr $\lambda z.z\mathrm{MN} = (\lambda p.p\ \mathrm{F})\ \lambda z.z\mathrm{MN}$

$\rightarrow_\beta (\lambda z.z\mathrm{MN})\ \mathrm{F}$

$\rightarrow_\beta \mathrm{FMN}$

$\rightarrow_\beta \mathrm{N}$

Null and **null?**

null ≡ $\lambda x.\mathbf{T}$

null? ≡ $\lambda x.(x\ \lambda y.\lambda z.\mathbf{F})$

null? null $\rightarrow \lambda x.(x\ \lambda y.\lambda z.\mathbf{F})\ (\lambda x.\ \mathbf{T})$

$\rightarrow_\beta (\lambda x.\ \mathbf{T})(\lambda y.\lambda z.\mathbf{F})$

$\rightarrow_\beta \mathbf{T}$

Null and **null?**

null ≡ $\lambda x.\mathbf{T}$

null? ≡ $\lambda x.(x\ \lambda y.\lambda z.\mathbf{F})$

null? (cons M N) $\rightarrow \lambda x.(x\ \lambda y.\lambda z.\mathbf{F})\ \lambda z.z\mathrm{MN}$

$\rightarrow_\beta (\lambda z.z\ \mathrm{MN})(\lambda y.\lambda z.\mathbf{F})$

$\rightarrow_\beta (\lambda y.\lambda z.\mathbf{F})\ \mathrm{MN}$

$\rightarrow_\beta \mathbf{F}$

Counting

0 ≡ null

1 ≡ **cons F 0**

2 ≡ **cons F 1**

3 ≡ **cons F 2**

…

succ ≡ $\lambda x.$**cons F** x

pred ≡ $\lambda x.$**cdr** x

---

42 = $\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

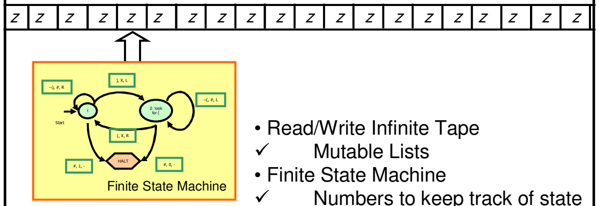
$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y$

$\lambda xy.(\lambda z.z\ xy)\ \lambda xy.\ y\ \lambda x.\mathrm{T}$

Lambda Calculus is a Universal Computer



- Read/Write Infinite Tape
  - ✓ Mutable Lists
- Finite State Machine
  - ✓ Numbers to keep track of state
- Processing
  - ✓ Way of making decisions (if)
  - ☯ Way to keep going

We have this, but we cheated using ≡ to make recursive definitions!

Way to Keep Going

$(\lambda\, f.\ ((\lambda\ x.f\,(xx))\ (\lambda\ x.\ f\,(xx))))\ (\lambda z.z)$

$\rightarrow_\beta\ (\lambda x.(\lambda z.z)(xx))\ (\lambda\ x.\ (\lambda z.z)(xx))$

$\rightarrow_\beta\ (\lambda z.z)\ (\lambda\ x.(\lambda z.z)(xx))\ (\lambda\ x.(\lambda z.z)(xx))$

$\rightarrow_\beta\ (\lambda x.(\lambda z.z)(xx))\ (\lambda\ x.(\lambda z.z)(xx))$

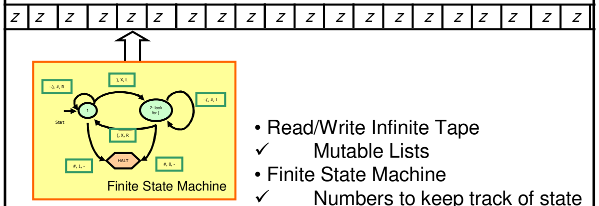
$\rightarrow_\beta\ (\lambda z.z)\ (\lambda\ x.(\lambda z.z)(xx))\ (\lambda\ x.(\lambda z.z)(xx))$

$\rightarrow_\beta\ (\lambda x.(\lambda z.z)($

$\rightarrow_\beta$ …

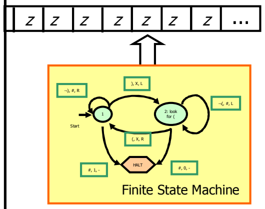
This should give you some belief that we might be able to do it. We won't cover the details of why this works in this class.

Lambda Calculus is a Universal Computer



- Read/Write Infinite Tape
  - ✓ Mutable Lists
- Finite State Machine
  - ✓ Numbers to keep track of state
- Processing
  - ✓ Way of making decisions (if)
  - ✓ Way to keep going

Equivalent Computers



Turing Machine

≡

term = *variable*
| *term term*
| (*term*)
| **λ** *variable* . *term*

$\lambda y.\ M \Rightarrow_\alpha \lambda v.\ (M\ [y\ \alpha\ \ v])$
where v does not occur in M.

$(\lambda x.\ M)N \Rightarrow_\beta M\ [\ x\ \alpha\ N\ ]$

Lambda Calculus

Universal Computer

- Lambda Calculus can simulate a Turing Machine
  - Everything a Turing Machine can compute, Lambda Calculus can compute also
- Turing Machine can simulate Lambda Calculus (we didn’t prove this)
  - Everything Lambda Calculus can compute, a Turing Machine can compute also
- Church-Turing Thesis: this is true for any other mechanical computer also

Charge

- Wednesday: Non-Deterministic Computing (P vs. NP question)
- Qualify for ps9 presentation by midnight Friday night