λ Calculus & Computability

Yan Huang
Dept. of Comp. Sci.
University of Virginia

Review of the Turing Machine

- Formalism $(Q, \Gamma, \Sigma, \delta, q_{start}, q_{accept}, q_{reject})$
- Abstract Problems
- Language Problems
- Computation
- Computability vs. Decidability

Today we are looking at a completely different formal computation model – the λ-Calculus!

Calculus

- What is calculus?
  - **Calculus** is a branch of mathematics that includes the study of limits, derivatives, integrals, and infinite series.
- Examples

$d(uv) = v(du) + u(dv)$ The product rule

$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}.$ The chain rule

Real Definition

- ***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, geometry, etc.

λ Calculus Formalism (Grammar)

- Key words: **λ . ( ) terminals**
- *term* → *variable*

  | (*term*)

  | λ *variable* . *term*

  | *term term*

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

λ Calculus Formalism (rules)

- Rules

α-reduction (renaming)

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

where v does not occur in M.

β-reduction (substitution)

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

Try Example 1, 2, & 3 on the notes now!

Replace all x's in M with N

Free and Bound variables

- In λ Calculus all *variable*s are local to function definitions
- Examples
  - $\lambda x.xy$
    x is bound, while y is free;
  - $(\lambda x.x)(\lambda y.yx)$
    x is bound in the first function, but free in the second function
  - $\lambda x.(\lambda y.yx)$
    x and y are both bound variables. (it can be abbreviated as $\lambda xy.yx$)

Be careful about β-Reduction

- $(\lambda x.\ M)N \Rightarrow M\ [\ x \mapsto N\ ]$

Try Example 4 on the notes now!

Replace all x's in M with N

If the substitution would bring a free variable of N in an expression where this variable occurs bound, we rename the bound variable before the substitution.

Computing Model for λ Calculus

- ***redex***: a *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*.

β-Reduction represents all the computation capability of Lambda calculus.

Another exercise

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

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

- 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

Alonzo Church (1903~1995)

Lambda Calculus

Church-Turing thesis

If an algorithm (a procedure that terminates) exists then there is an equivalent Turing Machine or applicable λ-function for that algorithm.

Alan M. Turing (1912~1954)

- Turing Machine
- Turing Test
- Head of Hut 8

Advisor:
Alonzo Church

Equivalence in Computability

- λ Calculus ↔ Turing Machine
  - (1) Everything computable by λ Calculus can be computed using the Turing Machine.
  - (2) Everything computable by the Turing Machine can be computed with λ Calculus.

Simulate λ Calculus with TM

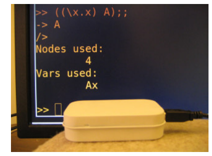
- The initial tape is filled with the initial λ *expression*
- Finite number of reduction rules can be implemented by the finite state automata in the Turing Machine
- Start the Turing Machine; it either stops – ending with the λ *expression* on tape in normal form, or continues forever – the β-reductions never ends.

WPI hacker implemented it on Z8 microcontroller

On Zilog Z8 Encore

λ Calculus in a Can

- Project LambdaCan



Refer to http://alum.wpi.edu/~tfraser/Software/Arduino/lambdacan.html for instructions to build your own λ-can!

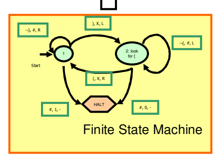
Equivalence in Computability

- λ Calculus ↔ Turing Machine
  - (1) Everything computable by λ Calculus can be computed using the Turing Machine.
  - (2) Everything computable by the Turing Machine can be computed with λ Calculus.

Simulate TM with λ Calculus

- Simulating the Universal Turing Machine

z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z



Finite State Machine

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

Making Decisions

- What does **decision** mean?
  - Choosing different ***strategies*** depending on the ***predicate***

if T $M\ N \rightarrow M$
if F $M\ N \rightarrow N$

- 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:

Finding the Truth

$\textbf{if} \equiv \lambda pca\ .\ pca$
$\textbf{T} \equiv \lambda xy.\ x$
$\textbf{F} \equiv \lambda xy.\ y$

$\textbf{if T}\ \boldsymbol{M\ N}$
$((\lambda pca\ .\ pca)\ (\lambda xy.\ x))\ \boldsymbol{M\ N}$
$\rightarrow_\beta (\lambda ca\ .\ (\lambda xy.\ x)\ ca))\ \boldsymbol{M\ N}$
$\rightarrow_\beta \rightarrow_\beta (\lambda xy.\ x)\ \boldsymbol{M\ N}$
$\rightarrow_\beta (\lambda y.\ \boldsymbol{M}\ ))\ \boldsymbol{N} \rightarrow_\beta \boldsymbol{M}$

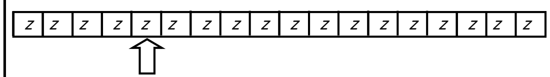
Try out reducing (if F T F) on your notes now!

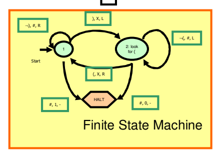
and and **or**?

- **and** ≡ $\lambda xy.$(**if** $x\ y$ **F**) **much more human-readable!**
  - $\rightarrow_\beta \lambda xy.((\lambda pca.pca)\ x\ y$ **F**)
  - $\rightarrow_\beta \lambda xy.(x\ y$ **F**)
  - $\rightarrow_\beta \lambda xy.(x\ y\ (\lambda uv.v))$
- **or** ≡ $\lambda xy.$(**if** x **T** y)

Simulate TM with λ Calculus

- Simulating the Universal Turing Machine





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

What is 11?

eleven **elf** **undici**

11 十一

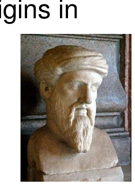
once **XI**

أحد عش **onze** **о д и н н а д ц а т ь**

イレブン

Numbers

- The natural numbers had their origins in the words used to count things
- Numbers as abstractions



pred (**succ** N) → N
succ (**pred** N) → N
pred (**0**) → **0**
succ (**zero**) → **1**

Defining Numbers

- In Church numerals, ***n*** is represented as a function that maps any function f to its n-fold composition.
- **0** ≡ $\lambda f\,x.\ x$
- **1** ≡ $\lambda f\,x.\ f\,(x)$
- **2** ≡ $\lambda f\,x.\ f\,(f\,(x))$

Defining **succ** and **pred**

- **succ** ≡ $\lambda\ n\,f\,x.\ f\,(n\,f\,x)$
- **pred** ≡ $\lambda\ n\,f\,x.\ n\ (\lambda gh.\ h\ (g\,f))\ (\lambda u.\ x)\ (\lambda u.\ u)$

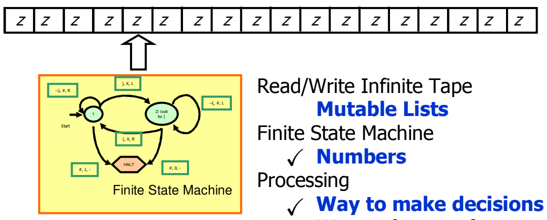
$$\mathrm{pred}(n) = \begin{cases} 0 & \text{if } n = 0, \\ n - 1 & \text{otherwise} \end{cases}$$

- **succ 1** $\rightarrow_\beta$ **?**

We'll see later how to deduce the term for **pred** using knowledge about **pairs**.

Simulate TM with λ Calculus

- Simulating the Universal Turing Machine



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

Defining List

- List is *either*
  - (1) **null**; *or*
  - (2) a **pair** whose second element is a list.

How to define **null** and **pair** then?

null, null?, pair, first, rest

null? null $\rightarrow$ **T**
null? (**pair** M N) $\rightarrow$ **F**

first (pair M N **)** $\rightarrow M$
rest (pair M N**)** $\rightarrow N$

null and null?

- **null** $\equiv \lambda x.\mathbf{T}$
- **null?** $\equiv \lambda x.(x\ \lambda yz.\mathbf{F})$

- **null? null** $\rightarrow_\beta (\lambda x.(x\ \lambda yz.\mathbf{F}))\ (\lambda x.\ \mathbf{T})$
  $\rightarrow_\beta (\lambda x.\ \mathbf{T})(\lambda yz.\mathbf{F})$
  $\rightarrow_\beta \mathbf{T}$

Defining Pair

- A pair [a, b] = (**pair** a b) is represented as $\lambda z.z\ a\ b$
- **first** $\equiv \lambda p.p\ \mathbf{T}$
- **rest** $\equiv \lambda p.p\ \mathbf{F}$
- **pair** $\equiv \lambda x\ y\ z.z\ x\ y$
- **first** (**cons** M N)

$\rightarrow_\beta$ ($\lambda p.p$ **T**) (**pair** M N)
$\rightarrow_\beta$ (**pair** M N) **T** $\rightarrow_\beta$ ($\lambda z.z$ M N) **T**
$\rightarrow_\beta$ **T** M N
$\rightarrow_\beta$ M

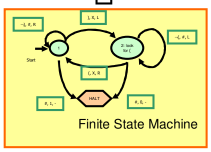
Defining **pred**

- **C** $\equiv \lambda pz.(z$ (**succ** (**first** p)) (**first** p))
  Obviously, **C [n, n-1]** $\rightarrow_\beta$ **[n+1, n]**, i.e., **C** turns a pair **[n, n-1]** to be **[n+1, n].**
- **pred** $\equiv$ **rest** $(\lambda n\ .\ n$ **C** $(\lambda z.z$ **0 0**$))$

Simulate TM with λ Calculus

- Simulating the Universal Turing Machine

z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z



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

Simulate Recursion

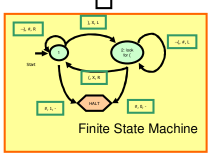
$$(\lambda f.\ ((\lambda\ x.f\ (xx))\ (\lambda\ x.\ f\ (xx))))\ (\lambda z.z)$$
$$\to_\beta\ (\lambda x.(\lambda z.z)(xx))\ (\lambda\ x.\ (\lambda z.z)(xx))$$
$$\to_\beta\ (\lambda z.z)\ (\lambda\ x.(\lambda z.z)(xx))\ (\lambda\ x.(\lambda z.z)(xx))$$
$$\to_\beta\ (\lambda x.(\lambda z.z)(xx))\ (\lambda\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

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.

Simulate TM with λ Calculus

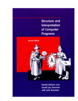
- Simulating the Universal Turing Machine

z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z	z



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

Introducing Scheme



- Scheme is a dialect of LISP programming language
- Computation in Scheme is a little higher level than in λ-Calculus in the sense that the more "human-readable" primitives (like **T**, **F**, **if**, **natural numbers**, **null**, **null?**, and **cons**, etc) have already been defined for you.
- The basic reduction rules are exactly the same.

A Turing simulator in Scheme

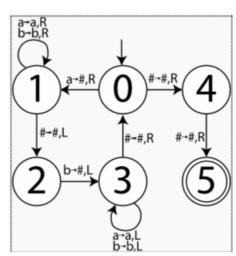
```scheme
(define trace-state
  (lambda (tape head-position blank-symbol)
    (letrec ((print-n-times (lambda (s n)
                              (when (> n 0)
                                (display s)
                                (print-n-times s (- n 1)))))
             (print-tape (lambda (tape n)
                           (when (not (or (null? tape)
                                          (= n 0)))
                             (display (car tape))
                             (print-tape (cdr tape) (- n 1))))))
      (when (< head-position trace-tape-chars)
        (print-n-times #\space head-position)
        (display "v\n"))
      (print-tape tape trace-tape-chars)
      (print-n-times blank-symbol (max 0 (- trace-tape-chars (length tape))))
      (display "\n"))))

(define find-trans-state
  (lambda (states state symbol blank-symbol)
    (if (null? states)
      (values 'invalid blank-symbol 'left)
      (if (equal? (list state symbol) (caar states))
        (let ((s (cadar states)))
          (values (list-ref s 0) (list-ref s 1) (list-ref s 2)))
        (find-trans-state (cdr states) state symbol blank-symbol)))))

(define simulate
  (lambda (tape state head-position transition-func accepting-states blank-symbol)
    (trace-state tape head-position blank-symbol)

    (cond ((eq? state 'invalid) #f)
          ((member state accepting-states) #t)
          (else (let ((symbol (symbol-at tape head-position blank-symbol)))
                  (let-values (((newstate newsymbol movedir) (transition-func state symbol)))
                    (simulate (write-to-tape tape head-position newsymbol)
                              newstate
                              (cond ((eq? movedir 'left)  (- head-position 1))
                                    ((eq? movedir 'right) (+ head-position 1))
                                    (else (error "Illegal head move")))
                              transition-func
                              accepting-states
                              blank-symbol)))))))
```


TM Simulator demonstration



A Turing Machine recognizing $a^n b^n$

```scheme
(define test-states
'(((0 \#)  (4 \# right))
  ((0 a)   (1 \# right))
  ((4 \#)  (5 \# right))
  ((1 a)   (1 a  right))
  ((1 b)   (1 b  right))
  ((1 \#)  (2 \# left))
  ((2 b)   (3 \# left))
  ((3 a)   (3 a  left))
  ((3 b)   (3 b  left))
  ((3 \#)  (0 \# right))))
```

Encoding of the FSM in Scheme.

Summary: TM and λ Calculus

- λ Calculus emphasizes the use of transformation rules and does not care about the actual machine implementing them.
- It is an approach more related to software than to hardware

Many slides and examples are adapted from materials developed for Univ. of Virginia CS150 by David Evans.