Introduction to the $\lambda$-Calculus

Part I

CS 209 - Functional Programming

Dr. Greg Lavender
Department of Computer Science
Stanford University

Historical Origins

- Foundations of Mathematics (1879-1936)
  - Paradoxes of set theory
    - Russell’s paradox: $\{ x \mid x \notin x \}$
  - Axiomatic systems, mathematical logic & type theory
    - Hilbert, Bernays, Brouwer, Russell, Zermelo, Fraenkel, etc.
  - Theory of computable functions
    - Church’s thesis tells us all of these are equivalent notations for representing the class of computable functions
      - Combinatory logic - Haskell Curry (1930)
      - $\lambda$-Calculus - Alonzo Church (1934)
      - Primitive recursive functions - Stephen Kleene (1936)
      - Turing machines - Alan Turing (1936)
Introduction to the $\lambda$-calculus

Alonzo Church

- PhD from Princeton University, 1927
  - Advisor was Oswald Veblen
    - who also advised R. L. Moore of UT Austin
  - Also studied under David Hilbert in Germany
    - Hilbert’s famous “23 problems” from 1900
      - e.g., Hilbert’s 10th problem solved in 1970 by Yuri Matiyasevich
    - also see Millennium Problems from 2000
      - e.g., P vs NP has a $1,000,000.00 prize
      - http://www.claymath.org/millennium/
  - Many of Church’s PhD students are considered “founders” of computer science
    - Stephen Kleene, 1934 (recursive function theory)
    - J. Barkley Rosser, 1934 (Church-Rosser theorem)
    - Alan Turing, 1938 (computational logic & computability)
    - Martin Davis, 1950 (logic & computability theory)
    - J. Hartley Rogers, 1952 (recursive function theory)
    - Michael Rabin, 1956 (probabilistic algorithms – Turing award)
    - Dana Scott, 1958 (prob. algorithms, domain theory - Turing award)
    - Raymond Smullyan, 1959 (logic, tableau proof, formal systems)

The $\lambda$-calculus in Computer Science

- A formal notation, theory, and model of computation
  - Church’s thesis: $\lambda$-calculus & Turing Machines describe the same set of objects, i.e., effectively computable functions
    - equivalence was proven by Kleene
- Untyped and typed $\lambda$-calculus
  - e.g., $\lambda x.x*x$ versus $\lambda x:\text{int}.x*x$
- Foundation for the functional style of programming
  - Lisp, Scheme, ISWIM, ML, Miranda™, O’Caml, Haskell
  - Peter Landin also used the lambda calculus to describe Algol-60, which is the father/mother of all modern procedural and OO languages used today: C, C++, Java, etc.
- Notation for Scott-Strachey denotational semantics
  - can be used to abstractly represent:
    - numbers, booleans, predicates, functions, variables, block scopes, expressions, ordered pairs, lists, records/structs & recursion
    - calling conventions: call-by-value, call-by-name
    - types, polymorphism, type inferencing
    - Curry-Howard Correspondence between proofs and types
A Computational Point of View

- A function is well-defined iff for every input there is at most one output
- a function in set theory is a graph
- characterized solely by an input->output relation
  - *extensional equality* - two functions $f$ & $g$ are equal iff they have the same graph,
    - i.e., $\{(x,y) \mid y = f(x)\}$
- this doesn’t work too well in programming!
  - in what way are two sorting functions equivalent?
    - bubblesort and quicksort produce the same graph, but are they equal computationally?
  - *intensional equality* - equivalent algorithmic complexity
  - how the function computes its result is important
  - in CS we also characterize a function by its algorithm:
    - e.g., a $O(n^2)$ vs $O(n \log_2 n)$ sorting algorithm

Some well-known functional notations

**Set Theoretic:**

$\{(x,y) \mid \forall x,y \in \mathbb{N} : y = x^2\}$

**Algebraic:**

$f : \mathbb{N} \rightarrow \mathbb{N}$

$f(x) = x^2$

**Untyped $\lambda$-notation:**

$(\lambda x.x \times x)$

**Typed $\lambda$-notation:**

$(\lambda x:\text{int}.x \times x)$

**Polymorphic $\lambda$-notation:**

$(\lambda x:\alpha.x \times x)$

**LISP:**

`(defun square(x) (* x x))`

**Scheme:**

`(define square (lambda (x) (* x x)))`

**Algol60:**

`integer procedure square(x); integer x; begin square := x * x end;`

**Pascal:**

`function square (x:integer) : integer; begin square := x * x end;`

**K&R C:**

`square(x) int x; { return (x * x); }`

**ANSI C/C++ & Java:**

`int square(int x) { return (x * x); }`

**ML97:**

`fun square x = x * x;`

`fun square (x:int) = x * x;`

`val square = fn x => x * x;`

**Haskell:**

`square :: Integer -> Integer`

`square x = x * x`

`map (\x -> x * x) [1,2,3,4,5]`

`[(x,y) \mid x \leftarrow [0..], y \leftarrow [x * x]]`
Definitions

- \( \lambda \)-calculus is a formal notation for defining functions
  - The \( \lambda \) operator acts as a “binding” operator that binds a variable and limits its scope to an expression
  - Expressions in this notation are called \( \lambda \)-expressions
  - Every \( \lambda \)-expression denotes a function
  - A \( \lambda \)-expression consists of 3 kinds of terms:
    - **Variables**: \( x, y, z, \) etc.
      - We use \( V, V_1, V_2, \) etc., for arbitrary variables
    - **Abstractions**: \( \lambda V.E \)
      - Where \( V \) is some variable and \( E \) is another \( \lambda \)-term
    - **Applications**: \( (E_1 E_2) \)
      - Where \( E_1 \) and \( E_2 \) are \( \lambda \)-terms
      - Applications are sometimes called combinations

The World’s Smallest Functional Programming Language

BNF definition of the lambda calculus:

\[
\begin{align*}
\langle \text{\lambda-term} \rangle & ::= \langle \text{variable} \rangle \\
& \quad \mid \lambda \langle \text{variable} \rangle . \langle \text{\lambda-term} \rangle \\
& \quad \mid (\langle \text{\lambda-term} \rangle \langle \text{\lambda-term} \rangle)
\end{align*}
\]

\[
\begin{align*}
\langle \text{variable} \rangle & ::= x \mid y \mid z \mid \ldots
\end{align*}
\]

Or, more compactly:

\[
\begin{align*}
E & ::= V \mid \lambda V.E \mid (E_1 E_2) \\
V & ::= x \mid y \mid z \mid \ldots
\end{align*}
\]

Where \( V \) is an arbitrary variable and \( E_i \) is an arbitrary \( \lambda \)-expression.

We call \( \lambda V \) the **head** of the \( \lambda \)-expression and \( E \) the **body**.
Variables

- variables can be bound or free
- the λ-calculus assumes an infinite universe of free variables
- they are bound to functions in an environment
- they become bound by usage in an abstraction
  - for example, in the λ-expression:
    $$\lambda x.x \cdot y$$
    $x$ is bound by $\lambda$ over the body $x \cdot y$, but $y$ is a free variable. I.e., lexically scoped. Compare this to the following in Scheme:

(broadcast z 3)
(broadcast x 2)
(broadcast y 2)
(broadcast multi-by-y (lambda (x) (* x y)))
(multi-by-y z) => 6

Abstractions

- if $\lambda V.E$ is an abstraction
  - $V$ is a bound variable over the body $E$
  - it denotes the function that when given an actual argument 'a', evaluates to the expression $E'$ with all occurrences of $V$ in $E$ replaced with 'a', written $E[a/V]$
  - For example the abstraction:
    $$\lambda x.x$$
    is the identity function
    $$(\lambda x.x)1 => 1$$
    $$(\lambda x.x)a => a$$
    $$(\lambda x.x)(\lambda x.x) => (\lambda x.x)$$
  - Compare to (lambda (x) x) in Scheme
Applications

- If $E_1$ and $E_2$ are $\lambda$-expressions, so is $(E_1 E_2)$
  - *application* is essentially function evaluation
  - apply the function $E_1$ to the argument $E_2$
    - $E_1$ is called the *rator* (operator)
    - $E_2$ is called the *rand* (operand)
  - Like in Scheme: $(E_1 E_2)$ such as
    - $((\text{lambda } (x) x) 1)$ where $E_1 = (\text{lambda } (x) x)$ and $E_2 = 1$
  - For example:
    - $(\lambda x. xx) 1 \Rightarrow 11$
    - $(\lambda x. xx) a \Rightarrow aa$
    - $(\lambda x. xx) (\lambda x. xx) \Rightarrow (\lambda x. xx) (\lambda x. xx) \Rightarrow ...$

  this last example doesn’t terminate. It keeps duplicating itself ad infinitum. In this example, we don’t care, but in real programming we do care about non-terminating evaluations.

  The expression $(\lambda x. xx) (\lambda x. xx)$ has a special name, we call it *Omega.*

Computation in the Lambda Calculus

- Computation occurs by converting two expressions in an application to another expression until we can no longer “reduce” the resulting expressions to a simpler form
- Reduction occurs by doing substitutions of arguments for parameters.
  - we can do either *eager evaluation* and reduce the arguments first then substitute them for parameters. This is called *applicative order.*
  - or we can do *lazy evaluation* and substitute argument expressions for parameters first, and then reduce them later. This is called *normal order.*
Conversion/reduction rules

- We need some computation "rules" that tell us how to do the substitutions:
  - \( \alpha \)-conversion
    - any abstraction \( \lambda V. E \) can be converted to
    - \( \lambda V. E[V/V] \) iff \([V/V]\) in \( E \) is valid
  - \( \beta \)-conversion (\( \beta \)-reduction)
    - any application \( (\lambda V. E_1) E_2 \) can be converted to
    - \( E_1[E_2/V] \) iff \([E_2/V]\) in \( E_1 \) is valid
  - \( \eta \)-conversion
    - any abstraction \( \lambda V. (E V) \) where \( V \) has no free occurrences in \( E \) can be converted to \( E \)

Conversion rule notation

\[
\begin{align*}
E_1 \xrightarrow{\alpha} E_2 & \quad \text{bound variable renaming to avoid naming conflicts} \\
E_1 \xrightarrow{\beta} E_2 & \quad \text{like a function call evaluation} \\
E_1 \xrightarrow{\eta} E_2 & \quad \text{elimination of irrelevant information}
\end{align*}
\]
**Introduction to the λ-calculus**

### α-conversion

α-conversion is bound variable renaming applied to an α-redex iff no naming conflicts

deex = reducible expression

\[ \lambda x. x \xrightarrow{\alpha} \lambda y. y \]  
\[ (\lambda x. x)[y/x] \]

\[ \lambda x. f \ x \xrightarrow{\alpha} \lambda y. f \ y \]  
\[ (\lambda x. f \ x)[y/x] \]

\[ \lambda x. \lambda y. x + y \xrightarrow{\alpha} \lambda y. \lambda y. f \ y + y \]  
not valid since y is already a bound variable in the body

### β-reduction

Computation through rewriting

\[ (\lambda x. f \ x) \ E \xrightarrow{\beta} f \ E \]

\[ (\lambda x. (\lambda y. x + y)) \ 3 \xrightarrow{\beta} \lambda y. 3 + y \]

\[ (\lambda y. 3 + y) \ 4 \xrightarrow{\beta} 3 + 4 \]
Evaluation order

- We can evaluate an expression left-to-right, innermost first
  - this is called "normal order" evaluation
  - $(\lambda y.1)((\lambda x.xx)(\lambda x.xx)) \Rightarrow 1$
  - lazy or non-strict evaluation like in Haskell
- Or we can evaluate right-to-left, innermost first
  - this is called "applicative order" evaluation
  - $(\lambda y.1)((\lambda x.xx)(\lambda x.xx)) \Rightarrow (\lambda y.1)((\lambda x.xx)(\lambda x.xx)) \Rightarrow \ldots$
    - never terminates!
  - eager or strict evaluation
    - if non-termination is a problem with eager evaluation, why is it the most common argument evaluation technique in modern programming languages?

Normal Form Properties

- if $E_1 \Rightarrow^* E_2$ then $E_2$ was obtained by a sequence of reduction steps starting with $E_1$
- If $E_2$ can not be further reduced by $\beta$ (or $\eta$) conversion, then $E_2$ is in normal form
- Normal form just means that we can do no further reduction
  - i.e., we have the "result" of the computation
- For example:
  - numerals are already in normal form
  - $(\lambda x.x) 0$ is not in normal form
    - reduces to 0
Curried Functions

- Named after Haskell Curry who used them in combinatory logic
  - first used by Moses Schönfinkel in the 1920s
  - in lambda calculus, functions are typically written as curried functions, i.e., unary functions
    - $\lambda x. \lambda y. x + y$
  - but we can also write them as $n$-ary functions:
    - $\lambda xy. x + y$
    - $\lambda xyz. x + y + z$
- any $n$-ary function can be replaced by a composition of $n$ unary functions.
  - $f(x, y) \Rightarrow (fx)y$
  - $f(x_1, x_2, \ldots x_n) \Rightarrow ((\ldots ((fx_1)x_2)\ldots)x_n)$

Compare to Let expressions in Scheme

Scheme has let and let* operators for establishing local bindings of variables to values over a lexically scoped block. In a let expression, all values in the list of let bindings are evaluated, and then bound to the local variables. In a let* expression, the values are evaluated and bound to the variables sequentially:

```
(define x 2) ;; global scope
(let ((x 3) (y x)) (* x y)) => 6  
(let*((x 3) (y x)) (* x y)) => 9 ;; like “let” in Haskell
```

Let expressions in functional programming languages are just “syntactic sugar” for lambda expressions. Let is a $n$-ary lambda expression and let* is a curried lambda expression:

```
((lambda (x y) (* x y)) 3 2) => 6
((lambda (x) ((lambda (y) (* x y)) 2)) 3) => 6
```
Lambda expressions in Scheme

- **Sum**
  - (define sum (lambda (x y) (+ x y)))
  - (sum 2 3) => 5

- **“Curried” version**
  - (define sum (lambda (x) (lambda (y) (+ x y))))
  - ((sum 2) 3) => 5
  - (let ((f (sum 2))) (f 3)) => 5

Lambda expressions in ML

- A lambda expression in ML is written as
  
  ```ml```
  fn <args> => <body>
  ```ml```
  val sum = fn x => fn y => x + y;
  val it = fn : int -> int -> int
  val sum 3 2;
  val it = 5 : int

- A “curried” version
  
  ```ml```
  fun sum x = fn y => x + y;
  val sum = fn : int -> int -> int
  val sum 3;
  val it = fn : int -> int
  val it 2;
  val it = 5 : int
  val (sum 3) 2;
  val it = 5 : int
  val sum 3 2;
  val it = 5 : int
Lambda expressions in Haskell

- note the strong syntactic similarity to the \(\lambda\)-calculus

\(<\text{arg}> \rightarrow \text{<body>}\)

succ = \(n \rightarrow n + 1\)
succ 0 ⇒ (\(n \rightarrow n + 1\)) 0 ⇒ 0 + 1 ⇒ 1

succ = (\(a \rightarrow b \rightarrow a + b\)) 1
succ 0 ⇒ ((\(a \rightarrow b \rightarrow a + b\)) 1) 0
⇒ (\(b \rightarrow 1 + b\)) 0
⇒ 1 + 0
⇒ 1

- but we usually prefer the syntactic sugar form

succ n = n + 1
succ 0 ⇒ 0 + 1 ⇒ 1
succ (succ 0) ⇒ succ (0+1) ⇒ succ 1 ⇒ 1+1 ⇒ 2

Map & fold using lambda expressions

- Map and \texttt{foldl} are higher-order functions that are similar across functional languages
- often take a lambda expression as the function argument
- note the order of the “don’t care” argument in the fold examples

Scheme
- \(\texttt{(map (lambda (x) (* x x)) ‘(1 2 3 4 5))} \Rightarrow (1 4 9 16 25)\)
- \(\texttt{(foldl (lambda (_) (+ 1 _)) 0 ‘(1 2 3 4 5))} \Rightarrow 5\)

ML
- \(\texttt{map (fn x => x * x) [1,2,3,4,5];} \Rightarrow [1,4,9,16,25]:\text{int list}\)
- \(\texttt{foldl (fn (_,x) => x + 1) 0 [1,2,3,4,5];} \Rightarrow 5 : \text{int}\)

Haskell
- \(\texttt{map (\(x \rightarrow x * x\)) [1,2,3,4,5] \Rightarrow [1,4,9,16,25]}\)
- \(\texttt{foldl (\(x \_ \rightarrow x + 1\)) 0 [1,2,3,4,5]} \Rightarrow 5\)
- \(\texttt{foldl (\(x \_ \rightarrow x + 1\)) 0 “Hello, world”} \Rightarrow 12\)
Introduction to the λ-calculus

Is the λ-calculus Turing complete?

- Can we represent the class of Turing computable functions?
  - booleans and conditional functions
  - numbers and arithmetic functions
  - data structures, such as ordered pairs, lists, etc.
  - recursive functions
- Yes, but it is syntactically tedious!
  - real programming languages use syntactic sugar for common lambda expressions to make it easier for the human
    - lambda expressions exist in Scheme, ML, Haskell, and even Python
  - λ-calculus is more suitable as an abstract model of a programming language rather than as a practical programming language
  - used as an intermediate language for a compiler
  - used to represent the denotational semantics of some languages
    - For example, see the appendix in the Revised Report on the Algorithmic Language Scheme (R5RS) for the denotational semantics of Scheme

Church Booleans

- We define boolean values and logical operators in the λ-calculus as functions:

  True = T = λt.λf.t = λtf.t
  False = F = λt.λf.f = λtf.f
  AND = λxy.xy(λtf.f) = λxy.xyF
  OR = λxy.x(λtf.f)y = λxy.xyT
  NEG = λx.x(λuv.v)(λab.a) = λx.xFT

- Example:

  NEG True = (λx.x(λuv.v)(λab.a))(λtf.t)
  => (λtf.t)(λuv.v)(λab.a)
  => (λuv.v)
  => λtf.f
  => False
Church Booleans

- Given **true** and **false** as \( \lambda \)-expressions:
  
  \[
  \text{True} = \lambda tf.t \\
  \text{False} = \lambda tf.f
  \]

- We then define a “conditional” \( \lambda \)-expression:
  
  \[
  \text{test} = \lambda c. \lambda x. \lambda y. c \, x \, y
  \]
  
  where \( \text{test} \, \text{True} \, v \, w = (\lambda c. \lambda x. \lambda y. c \, x \, y) \, \text{True} \, v \, w \)
  
  => \* \( \text{True} \, v \, w \)
  
  = (\lambda tf.t) \, v \, w
  
  => \* \( v \)

  what is the result of: \( \text{test} \, \text{False} \, v \, w \)

Church Numerals

- The natural numbers and arithmetic
  
  - Peano’s axioms
    - PA1: 0 is a number
    - PA2: if \( n \) is a number, \( \text{succ}(n) \) is a number
    - PA3: 0 is not the \( \text{succ} \) of any number
    - PA4: if \( \text{succ}(n) = \text{succ}(m) \), then \( n = m \)
    - PA5: Induction schema
      - if \( 0, n \in S \) and \( \text{succ}(n) \in S \), every number \( \in S \)
  
  - zero and the successor function:
    - \( 0, 1 = \text{succ}(0), 2 = \text{succ}(\text{succ}(0)), \ldots \)
  
  - in the \( \lambda \)-calculus, we only have functions, so we define the natural numbers as functions:
    - \( 0 = \lambda s.(\lambda z.z) \), but we will write this as \( \lambda sz.z \),
    - then the rest of the natural numbers can be defined as:
      - \( 1 = \lambda sz.s(z), 2 = \lambda sz.s(s(z)), 3 = \lambda sz.s(s(s(z))), \ldots \)
Successor function

- So how do we write a successor function?
  - \( S = \lambda wyx. y(wyx) \)
- Let's test it on zero using applicative order
  - HW: try it using normal order
- \( S0 = (\lambda wyx. y(wyx))(\lambda sz. z) \)
  - \( \Rightarrow \lambda yx. y((\lambda sz. z)yx) \)
  - \( \Rightarrow \lambda yx. y((\lambda z. z)x) \)
  - \( \Rightarrow \lambda yx. y(x) \)
  - \( \Rightarrow \lambda sz. s(z) \)
  - \( = 1 \)
- Note that \( \lambda yx. y(x) = \lambda sz. s(z) \) under \( \alpha \)-conversion
  - the variables names can be changed as needed to (carefully) avoid variable name conflicts
  - all we are doing is defining syntactic patterns and rules of rewriting that mimic the semantics of arithmetic

Homework Assignment

- Evaluate the following in the \( \lambda \)-calculus using the previous definitions for AND, OR, True and False. Apply appropriate step-by-step reductions:
  - AND True True \( \Rightarrow \) True
  - AND True False \( \Rightarrow \) False
  - OR True False \( \Rightarrow \) True
  - OR False False \( \Rightarrow \) False
Homework Assignment

- Evaluate the following $\lambda$-expressions where $S$ is the successor function:
  - $S_1$, $S_2$, $S_3$

- Addition in the $\lambda$-calculus
  - let $2S_3$ represent $2+3$
  - write out the $\lambda$-expression for $2S_3$
    - note that you will have to use $\alpha$-conversion to avoid name conflict
  - show that $2S_3$ reduces to $5$

- Multiplication is done using the expression:
  - $\lambda xyz.x(yz)$
  - show that $(\lambda x y z . x(yz)) \ 2 \ 2 \Rightarrow^* \lambda s . z . s(s(s(z)))) = 4$