Lambda Calculus
Outline

Introduction and history

Definition of lambda calculus
  Syntax and operational semantics
  Minutia of $\beta$-reduction
  Reduction strategies

Programming with lambda calculus
  Church encodings
  Recursion

De Bruijn indices
What is the lambda calculus?

A very simple, but Turing complete, programming language

- created before concept of programming language existed!
- helped to define what Turing complete means!

Lambda calculus syntax

\[ v \in \text{Var} ::= x \mid y \mid z \mid \ldots \]

\[ e \in \text{Exp} ::= v \quad \text{variable reference} \\
| e \; e \quad \text{application} \\
| \lambda v. \; e \quad \text{(lambda) abstraction} \]

Examples

\[
x \quad \lambda x. \; y \\
\lambda f. (\lambda x. \; f (x \; x)) \; (\lambda x. \; f (x \; x))
\]
Correspondence to Haskell

Lambda calculus is the **theoretical foundation** for **functional programming**

<table>
<thead>
<tr>
<th>Lambda calculus</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td>$f \ x$</td>
<td>$f \ x$</td>
</tr>
<tr>
<td>$\lambda x. \ x$</td>
<td>$\lambda x \to x$</td>
</tr>
<tr>
<td>$(\lambda f. \ f \ x) \ (\lambda y. \ y)$</td>
<td>$(\lambda f \to f \ x) \ (\lambda y \to y)$</td>
</tr>
</tbody>
</table>

Similar to Haskell with only: variables, application, anonymous functions

- amazingly, we don’t lose anything by omitting all of the other features!
  (for a particular definition of “anything”)
Early history of the lambda calculus

Origin of the lambda calculus:

- **Alonzo Church** in 1936, to formalize “computable function”
- proves Hilbert’s *Entscheidungsproblem* undecidable
  - provide an algorithm to decide truth of arbitrary propositions

Meanwhile, in England …

- young **Alan Turing** invents the Turing machine
- devises *halting problem* and proves undecidable

Turing heads to Princeton, studies under Church

- prove lambda calculus, Turing machine, general recursion are equivalent
- **Church–Turing thesis**: these capture all that can be computed
Why lambda?

Evolution of notation for a **bound variable**:

- Whitehead and Russell, *Principia Mathematica*, 1910
  - $2\hat{x} + 3$ – corresponds to $f(x) = 2x + 3$

- Church’s early handwritten papers
  - $\hat{x}. 2x + 3$ – makes scope of variable explicit

- Typesetter #1
  - $^x. 2x + 3$ – couldn’t typeset the circumflex!

- Typesetter #2
  - $\lambda x. 2x + 3$ – picked a prettier symbol

Impact of the lambda calculus

**Turing machine**: theoretical foundation for **imperative languages**
- Fortran, Pascal, C, C++, C#, Java, Python, Ruby, JavaScript, …

**Lambda calculus**: theoretical foundation for **functional languages**
- Lisp, ML, Haskell, OCaml, Scheme/Racket, Clojure, F#, Coq, …

In **programming languages research**:
- common language of discourse, formal foundation
- starting point for new features
  - extend syntax, type system, semantics
  - reveals precise impact and utility of feature
Outline

Introduction and history

**Definition of lambda calculus**
- Syntax and operational semantics
- Minutia of $\beta$-reduction
- Reduction strategies

**Programming with lambda calculus**
- Church encodings
- Recursion

De Bruijn indices
Syntax

Lambda calculus syntax

\[ v \in \text{Var} ::= \ x \mid \ y \mid \ z \mid \ldots \]

\[ e \in \text{Exp} ::= v \quad \text{variable reference} \]

\[ \quad \mid \ e \ e \quad \text{application} \]

\[ \quad \mid \ \lambda v. e \quad \text{(lambda) abstraction} \]

Abstraction extend as far right as possible
so … \[ \lambda x. x \ y \equiv \lambda x. (x \ y) \]
NOT \[ (\lambda x. x)y \]

Syntactic sugar

Multi-parameter functions:

\[ \lambda x. (\lambda y. e) \equiv \lambda x \ y. e \]
\[ \lambda x. (\lambda y. (\lambda z. e)) \equiv \lambda x \ y \ z. e \]

Application is left-associative:

\[ (e_1 \ e_2) \ e_3 \equiv e_1 \ e_2 \ e_3 \]
\[ ((e_1 \ e_2) \ e_3) \ e_4 \equiv e_1 \ e_2 \ e_3 \ e_4 \]
\[ e_1 \ (e_2 \ e_3) \equiv e_1 \ (e_2 \ e_3) \]

Definition of lambda calculus
**β-reduction: basic idea**

A **redex** is an expression of the form: \((\lambda v. e_1) \ e_2\)

(an application with an abstraction on left)

Reduce by **substituting** \(e_2\) for every reference to \(v\) in \(e_1\)

write this as: \([e_2/v]e_1\)

Lots of different notations for this!

**Simple example**

\((\lambda x. x \ y \ x) \ z \mapsto z \ y \ z\)
Operational semantics

\[ e \in \text{Exp} ::= \upsilon \mid e \mid \lambda \upsilon. e \]

Reduction semantics

\[ (\lambda \upsilon. e_1) e_2 \mapsto [e_2/\upsilon]e_1 \]

\[ \frac{e \mapsto e'}{\lambda \upsilon. e \mapsto \lambda \upsilon. e'} \]

\[ \frac{e_1 \mapsto e_1'}{\frac{e_1 e_2 \mapsto e_1' e_2}{e_1 e_2 \mapsto e_1' e_2}} \quad \frac{e_2 \mapsto e_2'}{\frac{e_1 e_2 \mapsto e_1 e_2'}{e_1 e_2 \mapsto e_1' e_2'}} \]

Note: Reduction order is ambiguous!
Exercise

Apply \( \beta \)-reduction in the following expressions

Round 1:

- \((\lambda x. x)\) \(z\)
- \((\lambda x y. x)\) \(z\)
- \((\lambda x y. x)\) \(z u\)

Round 2:

- \((\lambda x. x x)\) \((\lambda y. y)\)
- \((\lambda x. (\lambda y. y)\) \(z)\)
- \((\lambda x. (x (\lambda y. x)))\) \(z\)

\[
e \in \text{Exp} \ ::= \ v \mid e \ e \mid \lambda v. e
\]

\[
(\lambda v. e_1) \ e_2 \mapsto [e_2/v]e_1 \quad \frac{e \mapsto e'}{\lambda v. e \mapsto \lambda v. e'}
\]

\[
\frac{e_1 \mapsto e'_1}{e_1 \ e_2 \mapsto e'_1 \ e_2} \quad \frac{e_2 \mapsto e'_2}{e_1 \ e_2 \mapsto e_1 \ e'_2}
\]

Definition of lambda calculus
Outline

Introduction and history

Definition of lambda calculus
  Syntax and operational semantics
  Minutia of $\beta$-reduction
  Reduction strategies

Programming with lambda calculus
  Church encodings
  Recursion

De Bruijn indices
Variable scoping

An abstraction consists of:

1. a variable declaration
2. a function body – the variable can be referenced in here

The scope of a declaration: the parts of a program where it can be referenced

A reference is bound by its innermost declaration

Mini-exercise: 

\[(\lambda x. e_1 \ (\lambda y. e_2 \ (\lambda x. e_3))) \ (\lambda z. e_4)\]

• What is the scope of each variable declaration?
Free and bound variables

A variable $v$ is **free** in $e$ if:

- $v$ is referenced in $e$
- the reference is *not* enclosed in an abstraction declaring $v$ (within $e$)

If $v$ is referenced and enclosed in such an abstraction, it is **bound**

**Closed expression**: an expression with no free variables
- equivalently, an expression where all variables are bound
Exercise

1. Define the abstract syntax of lambda calculus as a Haskell data type

\[ e \in Exp ::= v \mid e \ e \mid \lambda v. e \]

2. Define a function: \textbf{free} :: Exp -> Set Var
   the set of free variables in an expression

3. Define a function: \textbf{closed} :: Exp -> Bool
   no free variables in an expression
Potential problem: variable capture

Principles of variable bindings:

1. variables should be bound according to their **static scope**
   - \( \lambda x. (\lambda y. (\lambda x. y \ x)) \ x \mapsto \lambda x. \lambda x. x \ x \)

2. how we name bound variables doesn’t really matter
   - \( \lambda x. x \equiv \lambda y. y \equiv \lambda z. z \) (\(\alpha\)-equivalence)

If violated, we can’t reason about functions separately from their use!

Example with naive substitution

A function that always returns its first argument: \( \lambda x y. x \) ... or does it?

\[
(\lambda x y. x) \ y \ u \mapsto (\lambda y. y) \ u \mapsto u
\]
Solution: capture-avoiding substitution

Capture-avoiding (safe) substitution: \([e/v]e'\)

\[
\begin{align*}
[e/v]v &= e \\
[e/v]w &= w & v \neq w \\
[e/v](e_1 e_2) &= [e/v]e_1 [e/v]e_2 \\
[e/v](\lambda u. e') &= \lambda w. [e/v][w/u]e' & w \notin \{v\} \cup FV(\lambda u. e') \cup FV(e)
\end{align*}
\]

\(FV(e)\) is the set of all free variables in \(e\)

Example with safe substitution

\[(\lambda x y. x) \ y \ u\]

\[
\begin{align*}
\rightarrow [y/x](\lambda y. x) u &= (\lambda z. [y/x][z/y]x) u = (\lambda z. [y/x]x) u = (\lambda z. y) u \\
\rightarrow [u/z]y &= y
\end{align*}
\]
Example

Recall example: \( \lambda x. (\lambda y. (\lambda x. y) x) \) \( \mapsto \lambda x. \lambda x. x \) x

Reduction with safe substitution

\[
\begin{align*}
\lambda x. (\lambda y. (\lambda x. y) x) & \mapsto \lambda x. [x/y](\lambda x. y) x = \lambda x. \lambda z. [x/y][z/x](y) x = \lambda x. \lambda z. [x/y](y) z \\
& = \lambda x. \lambda z. x z
\end{align*}
\]

Definition of lambda calculus
Outline

Introduction and history

Definition of lambda calculus
  Syntax and operational semantics
  Minutia of $\beta$-reduction
  Reduction strategies

Programming with lambda calculus
  Church encodings
  Recursion

De Bruijn indices
Question: what is a **value** in the lambda calculus?
- how do we know when we’re done reducing?

One answer: a value is an expression that **contains no redexes**
- called **$\beta$-normal form**

Not all expressions can be reduced to a value!

\[
(\lambda x.x)x \ (\lambda x.x) \ \mapsto \ (\lambda x.x) \ (\lambda x.x) \ \mapsto \ (\lambda x.x) \ (\lambda x.x) \ \mapsto \ldots
\]
Does reduction order matter?

Recall: operational semantics is ambiguous

- in what order should we $\beta$-reduce redexes?
- does it matter?

\[
e \mapsto e' \subseteq \text{Exp} \times \text{Exp}
\]

\[
(\lambda v. e_1) e_2 \mapsto [e_2/v]e_1
\]

\[
\begin{align*}
\lambda v. e &\mapsto \lambda v. e' \\
 e &\mapsto e' \\
 e_1 &\mapsto e'_1 \\
 e_2 &\mapsto e'_2 \\
 e_1 e_2 &\mapsto e'_1 e'_2
\end{align*}
\]

\[
e \mapsto^* e' \subseteq \text{Exp} \times \text{Exp}
\]

\[
\begin{align*}
s &\mapsto^* s \\
s &\mapsto s' \\
s' &\mapsto^* s'' \\
s &\mapsto^* s''
\end{align*}
\]
Church–Rosser Theorem

Reduction is **confluent**

If $e \rightarrow^* e_1$ and $e \rightarrow^* e_2$, then

$\exists e'$ such that $e_1 \rightarrow^* e'$ and $e_2 \rightarrow^* e'$

**Corollary**: any expression has **at most one normal form**

- if it exists, we can still reach it after any sequence of reductions
- … but if we pick badly, we might never get there!

Example: $(\lambda x. y) \ ( (\lambda x. x x) \ (\lambda x. x x))$
Reduction strategies

**Redex positions**

- **leftmost redex**: the redex with the leftmost $\lambda$
- **outermost redex**: any redex that is not part of another redex
- **innermost redex**: any redex that does not contain another redex

**Label redexes**

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

**Reduction strategies**

- **normal order reduction**: reduce the leftmost redex
- **applicative order reduction**: reduce the leftmost of the innermost redexes

**Compare reductions**: $$(\lambda x. y) ( (\lambda x. x x) \ (\lambda x. x x))$$
Exercises

Write **two reduction sequences** for each of the following expressions

- one corresponding to a normal order reduction
- one corresponding to an applicative order reduction

1. \((\lambda x. x x) ((\lambda x. y x) z (\lambda x. x))\)
2. \((\lambda x y z. x z) (\lambda z. z) ((\lambda y. y) (\lambda z. z)) x\)
Comparison of reduction strategies

**Theorem**
If a normal form exists, normal order reduction will find it!

**Applicative order:** reduces arguments first
- evaluates every argument exactly once, even if it's not needed
- corresponds to “call by value” parameter passing scheme

**Normal order:** copies arguments first
- doesn’t evaluate unused arguments, but may re-evaluate each one many times
- guaranteed to reduce to normal form, if possible
- corresponds to “call by name” parameter passing scheme
Lazy evaluation: reduces arguments only if used, but at most once
- essentially, an efficient implementation of normal order reduction
- only evaluates to “weak head normal form”
- corresponds to “call by need” parameter passing scheme

Expression $e$ is in **weak head normal form** if:
- $e$ is a variable or lambda abstraction
- $e$ is an application with a variable in the left position
... in other words, $e$ does not start with a redex
Outline

Introduction and history

Definition of lambda calculus
  Syntax and operational semantics
  Minutia of $\beta$-reduction
  Reduction strategies

Programming with lambda calculus
  Church encodings
  Recursion

De Bruijn indices
Data and operations are encoded as **functions** in the lambda calculus.

For Booleans, need lambda calculus terms for *true*, *false*, and *if*, where:

- \( \text{if true } e_1 e_2 \mapsto^* e_1 \)
- \( \text{if false } e_1 e_2 \mapsto^* e_2 \)

### Church Booleans

<table>
<thead>
<tr>
<th>True</th>
<th>False</th>
<th>If</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda x y. x )</td>
<td>( \lambda x y. y )</td>
<td>( \lambda b t e. b t e )</td>
</tr>
</tbody>
</table>

### More Boolean operations

<table>
<thead>
<tr>
<th>And</th>
<th>Or</th>
<th>Not</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda p q. \text{if } p q p )</td>
<td>( \lambda p q. \text{if } p p q )</td>
<td>( \lambda p. \text{if } p \text{ false true} )</td>
</tr>
</tbody>
</table>
Church numerals

A natural number $n$ is encoded as a function that applies $f$ to $x$ $n$ times

### Church numerals

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>$\lambda f. x. x$</td>
</tr>
<tr>
<td>one</td>
<td>$\lambda f. x. f \ x$</td>
</tr>
<tr>
<td>two</td>
<td>$\lambda f. x. f \ (f \ x)$</td>
</tr>
<tr>
<td>three</td>
<td>$\lambda f. x. f \ (f \ (f \ x))$</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$n$</td>
<td>$\lambda f. x. f^n \ x$</td>
</tr>
</tbody>
</table>

### Operations on Church numerals

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>succ</td>
<td>$\lambda n. f. x. (n \ f \ x)$</td>
</tr>
<tr>
<td>add</td>
<td>$\lambda n. m. f. x. n \ f \ (m \ f \ x)$</td>
</tr>
<tr>
<td>mult</td>
<td>$\lambda n. m. f. n \ (m \ f)$</td>
</tr>
<tr>
<td>isZero</td>
<td>$\lambda n. \ n \ (\lambda x. \ false) \ true$</td>
</tr>
</tbody>
</table>
Encoding values of more complicated data types

At a minimum, need **functions** that encode how to:

- **construct** new values of the data type
- **destruct and use** values of the data type in a general way

Can encode values of many data types as **sums of products**

Recall from domain theory

\[
\text{data } \text{Val} = \text{N Nat | B Bool | P (Nat,Bool)} \\
\equiv \\
\text{Nat } \oplus \text{Bool } \oplus (\text{Nat } \otimes \text{Bool})
\]
Products (a.k.a. tuples)

A tuple is defined by:
- a tupling function; last argument selects an earlier argument (constructor)
- a set of selecting functions (destructors)

### Church pairs

<table>
<thead>
<tr>
<th>Function</th>
<th>Lambda Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>pair</td>
<td>( \lambda x , y , s , s , x , y )</td>
</tr>
<tr>
<td>fst</td>
<td>( \lambda t \cdot t , (\lambda x , y \cdot x) )</td>
</tr>
<tr>
<td>snd</td>
<td>( \lambda t \cdot t , (\lambda x , y \cdot y) )</td>
</tr>
</tbody>
</table>

### Church triples

<table>
<thead>
<tr>
<th>Function</th>
<th>Lambda Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{tuple}_3 )</td>
<td>( \lambda x , y , z , s , s , x , y , z )</td>
</tr>
<tr>
<td>( \text{sel}_{1/3} )</td>
<td>( \lambda t \cdot t , (\lambda x , y , z \cdot x) )</td>
</tr>
<tr>
<td>( \text{sel}_{2/3} )</td>
<td>( \lambda t \cdot t , (\lambda x , y , z \cdot y) )</td>
</tr>
<tr>
<td>( \text{sel}_{3/3} )</td>
<td>( \lambda t \cdot t , (\lambda x , y , z \cdot z) )</td>
</tr>
</tbody>
</table>
Sums (a.k.a. tagged unions)

A tagged union is defined by:

- a case function: a tuple of functions (destructor)
- a set of tags that select the correct function and apply it (constructors)

Church either

\[
either = \lambda f g u.\ u f g
\]

\[
in_L = \lambda x f g.\ f x
\]

\[
in_R = \lambda y f g.\ g y
\]

Church union

\[
case_3 = \lambda f g h u.\ u f g h
\]

\[
in_{1/3} = \lambda x f g h.\ f x
\]

\[
in_{2/3} = \lambda y f g h.\ g y
\]

\[
in_{3/3} = \lambda z f g h.\ h z
\]
Outline

Introduction and history

Definition of lambda calculus
  Syntax and operational semantics
  Minutia of $\beta$-reduction
  Reduction strategies

Programming with lambda calculus
  Church encodings
  Recursion

De Bruijn indices
Naming in lambda calculus

Observation: can use abstractions to define names

\[
\text{let } \text{succ} = \lambda n \rightarrow n+1 \\
\text{in } \ldots \text{ succ } 3 \ldots \text{ succ } 7 \ldots
\]

\[
(\lambda \text{succ.} \\
\quad \ldots \text{ succ } 3 \ldots \text{ succ } 7 \ldots \\
\quad ) (\lambda n f x. f (n f x))
\]

But this pattern doesn’t work for recursive functions!

\[
\text{let } \text{fac} = \lambda n \rightarrow \\
\quad \ldots n \ast \text{ fac } (n-1) \\
\text{in } \ldots \text{ fac } 5 \ldots \text{ fac } 8 \ldots
\]

\[
(\lambda \text{fac.} \\
\quad \ldots \text{ fac } 5 \ldots \text{ fac } 8 \ldots \\
\quad ) (\lambda n f x. \ldots \text{ mult } n (??? (\text{pred } n)))
\]
Recursion via fixpoints

Solution: Fixpoint function

\[ Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \]

\[ Y \, g \]
\[ \mapsto (\lambda x. g (x x)) (\lambda x. g (x x)) \]
\[ \mapsto g ((\lambda x. g (x x)) (\lambda x. g (x x))) \]
\[ \mapsto g (g ((\lambda x. g (x x)) (\lambda x. g (x x)))) \]
\[ \mapsto g (g (g ((\lambda x. g (x x)) (\lambda x. g (x x))))) \]
\[ \mapsto ... \]

Example recursive function (factorial)

\[ Y (\lambda \text{fac} \, n. \, \text{if} \, \text{(isZero} \, n) \, \text{one} \, \text{(mult} \, n \, (\text{fac} \, (\text{pred} \, n))))) \]
Outline

Introduction and history

Definition of lambda calculus
  Syntax and operational semantics
  Minutia of $\beta$-reduction
  Reduction strategies

Programming with lambda calculus
  Church encodings
  Recursion

De Bruijn indices
The role of names in lambda calculus

Variable names are a convenience for readability (mnemonics) … but they’re annoying in implementations and proofs

Annoyances related to names

- safe substitution is complicated, requires generating fresh names (minor)
- checking and maintaining $\alpha$-equivalence is complicated and expensive (major)

Recall: $\alpha$-equivalence

Expressions are the same up to variable renaming

- $\lambda x. x \equiv \lambda y. y \equiv \lambda z. z$
- $\lambda x y. x \equiv \lambda y x. y$
A nameless representation of lambda calculus

Basic idea: de Bruijn indices

- an abstraction implicitly declares its input (no variable name)
- a variable reference is a number $n$, called a **de Bruijn index**, that refers to the $n$th abstraction up the AST

Nameless lambda calculus

$$
\begin{align*}
  n \in \text{Nat} &::= (\text{any natural number}) \\
  e \in \text{Exp} &::= e \ e \quad \text{application} \\
  &| \quad \lambda \ e \quad \text{lambda abstraction} \\
  &| \quad n \quad \text{de Bruijn index}
\end{align*}
$$

Named $\leadsto$ nameless

- $\lambda x. \ x \leadsto \lambda \ 0$
- $\lambda x \ y. \ x \leadsto \lambda \ \lambda \ 1$
- $\lambda x \ y. \ y \leadsto \lambda \ \lambda \ 0$
- $\lambda x. \ (\lambda y. \ y) \ x \leadsto \lambda \ (\lambda \ 0) \ 0$

Main advantage: $\alpha$-equivalence is just syntactic equality!
Deciphering de Bruijn indices

**De Bruijn index**: the number of $\lambda$s you have to *skip* when moving up the AST

```latex
def \debruijn (\lambda 0 (\lambda 1 (\lambda 0 12) 0)
\lambda x. x (\lambda y. x (\lambda z. z y x) y)
```

Gotchas:
- the same variable will be a different number in different contexts
- scopes work the same as before; references respect the AST
  - e.g. the blue $\theta$ refers to the blue $\lambda$ since it is not in scope of the green $\lambda$,
    and the green $\lambda$ does not count as a *skip*
Free variable in $e$: a de Bruijn index that skips over all of the $\lambda$s in $e$

- the same free variables will have the same number of $\lambda$s left to skip