History

• The lambda calculus was one of several computational systems defined by mathematicians to probe the foundations of logic
  • Others: combinator calculus, Turing machines

• Lambda calculus was introduced by Alonzo Church in the 1930’s
  • Originally used to establish the existence of an undecidable problem
A Language of Functions

• Like SKI calculus, lambda calculus focuses exclusively on functions as the essence of computation

\[ e \rightarrow x \mid \lambda x.e \mid e \ e \]

In words, a lambda expression is a

*variable* \( x \),

*an abstraction* (a function definition) \( \lambda x.e \), or

*an application* (a function call) \( e_1 \ e_2 \)
Computation Rule

\[(\lambda x. e_1) \, e_2 \rightarrow e_1 [x := e_2]\]

In words: In a function call, the \textit{formal parameter} \(x\) is replaced by the \textit{actual argument} \(e_2\) in the body of the function \(e_1\).

This is called \textit{beta reduction}. 
Examples

• The identity function $I$: $\lambda x. x$

• The constant function $K$: $\lambda z. \lambda y. z$

$$(\lambda x. x) \ (\lambda z. \lambda y. z) \to x \ [x := \lambda z. \lambda y. z] = \lambda z. \lambda y. z$$

$$((\lambda z. \lambda y. z) \ (\lambda x. x)) \ (\lambda a. \lambda b. a) \to (\lambda y. \ (\lambda x. x)) \ (\lambda a. \lambda b. a) \to \lambda x. x$$
Substitution

- Beta-reduction is the workhorse rule in the lambda calculus
  - But it relies on substitution

\[
x [x := e] = e \\
y [x := e] = y \\
(e_1 e_2) [x := e] = (e_1 [x := e]) (e_2 [x := e]) \\
(\lambda x. e_1) [x := e] = \lambda x. e_1 \\
(\lambda y. e_1) [x := e] = \lambda y. (e_1 [x := e]) \text{ if } x \neq y \text{ and } y \text{ does not appear free in } e
\]
Huh?

Why do we need this complicated rule?

$$\lambda y. e_1 \ [x := e] = \lambda y. (e_1 \ [x := e]) \text{ if } x \neq y \text{ and } y \text{ does not appear free in } e$$

Consider

$$\lambda y. x \ [x := y]$$
Free Variables

The *free variables* of an expression are the variables not bound in an abstraction.

\[
\begin{align*}
FV(x) &= \{ x \} \\
FV(e_1 e_2) &= FV(e_1) \cup FV(e_2) \\
FV(\lambda x.e) &= FV(e) - \{ x \}
\end{align*}
\]
Substitution Revisited

\[ x [x := e] = e \]
\[ y [x := e] = y \]
\[ (e_1 e_2) [x := e] = (e_1 [x := e]) (e_2 [x := e]) \]
\[ (\lambda x. e_1) [x := e] = \lambda x. e_1 \]
\[ (\lambda y. e_1) [x := e] = \lambda y. (e_1 [x := e]) \text{ if } x \neq y \text{ and } y \notin \text{FV}(e) \]
But Substitution Should Always Work ... 

• Intuitively, the bound variable name in an abstraction doesn’t matter
  • $\lambda x.x$ is as good as $\lambda y.y$

• We can rename bound variables to avoid collisions:

$$ (\lambda y.e_1) [x := e] = \lambda z.((e_1[y := z]) [x := e]) \text{ if } x \neq y \text{ and } z \text{ is a fresh name} $$

($fresh$ means not occurring in $e_1$ or $e$)
Revisiting Our Substitution Example ...

\((\lambda y. x) \ [ \ x := y \ ]\) =

\((\lambda z. x) \ [ \ x := y \ ]\) =

\((\lambda z. y)\)
Rules Again

• Renaming of bound variables is called *alpha conversion*

• Presentations of lambda calculus often include alpha conversion as a separate rule

• A third rule, *eta-conversion*, is also part of the lambda calculus but is not needed for computation:

\[ e = \lambda x. e \quad x \notin FV(e) \]
Summary

Lambda calculus has three rules:

• **Beta reduction** \((\lambda x. e_1) \ e_2 \rightarrow e_1 [x := e_2]\)

• **Alpha conversion** \(\lambda x. e = \lambda z. e [x := z]\) where \(z\) is fresh

• **Eta conversion** \(\lambda x. e \ x = e\) \(x \notin \text{FV}(e)\)

Lambda calculus is often presented emphasizing only beta reduction, with alpha conversion assumed to be done where needed to avoid capture of free variables (“capture-avoiding renaming”). Eta conversion is used mostly in proofs of logical properties, not in direct computation.
Example

$$(\lambda x. x x) (\lambda x. x x) \rightarrow x x [x := \lambda x. x x] = (\lambda x. x x) (\lambda x. x x)$$

• An example of a non-terminating expression
  • Reduces to itself in one step, so can always be reduced
Recursion

As with SKI, producing true recursion is just slightly more involved:

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

\[ Y g a = \lambda f. (\lambda x. f(x x)) (\lambda x. f(x x)) g a \rightarrow \]
\[ (\lambda x. g(x x)) (\lambda x. g(x x)) a \rightarrow \]
\[ g((\lambda x. g(x x)) (\lambda x. g(x x))) a \rightarrow \]
\[ g(g((\lambda x. g(x x)) (\lambda x. g(x x)))) a \rightarrow \]
\[ ... \]
Booleans

• As with SKI, represent true (false) by a function that given two arguments picks the first (second)

• True = K = \lambda x. \lambda y. x
• False = \lambda x. \lambda y. y
Boolean Operations

• Exactly like the SKI encoding ...

• Let B be a Boolean (T or F)

• $\text{Not}(B) = B \ F \ T$

• $B_1 \ OR \ B_2 = B_1 \ T \ B_2$

• $B_1 \ AND \ B_2 = B_1 \ B_2 \ F$
Integers

• \( N \) applies its first argument \( N \) times to its second argument

\[
n \ f \ x = f^n(x)
\]

\[
0 \ f \ x = x \quad \text{so} \quad 0 = \lambda f.\lambda x.x
\]

\[
i \ \text{inc} \ n \ f \ x = f \ (n \ f \ x) \quad \text{inc} = \lambda n.\lambda f.\lambda x. \ f \ (n \ f \ x)
\]
Factorial

one = inc 0
add = \( \lambda m. \lambda n. m \text{ inc } n \)
mul = \( \lambda m. \lambda n. m \text{ (add } n) \text{ 0} \)

pair = \( \lambda a. \lambda b. \lambda f. f a b \)
fst = \( \lambda p. p \lambda x. \lambda y. x \)
snd = \( \lambda p. p \lambda x. \lambda y. y \)

P = \( \lambda p. \text{ pair } (\text{ inc } (\text{ fst } p)) \text{ (mul } (\text{ fst } p) \text{ (snd } p)) \)
! = \( \lambda n. \text{ snd } (n \text{ P } (\text{ pair } \text{ one } \text{ one})) \)
Discussion

• The lambda calculus is extremely well-studied
  • More studied than combinator systems

• Some highlights:
  • General vs. primitive recursion
  • Confluence
  • Call-by-name vs. call-by-value
  • Abstract data types
Primitive Recursion (Again)

• This definition of factorial is not the textbook one
  • We didn’t use the Y combinator – we didn’t use general recursion

• Because we don’t need general recursion to define factorial

• Factorial is another example of a primitive recursive function
  • We use the iteration built into the definition of integers
Confluence

• The lambda calculus is confluent
  • The Church-Rosser theorem

• If $e_0 \rightarrow^* e_1$ and $e_0 \rightarrow^* e_2$, then there is an $e_3$ s.t. $e_1 \rightarrow^* e_3$ and $e_2 \rightarrow^* e_3$
  • Where we consider terms equivalent up to alpha conversion
Call-by-...

Given a redex

$$(\lambda x. e) \ e'$$

should we:

- Evaluate $e'$ before performing the beta reduction? \textit{call-by-value}
- Perform the beta reduction first? \textit{call-by-name}
Answers

• Answer 1: It mostly doesn’t matter, because of confluence

• Answer 2: For efficiency, call-by-value is better

• Answer 3: For termination, call-by-name is better
  • Call-by-name is guaranteed to terminate, if termination is possible
  • Call-by-value may fail to terminate even if call-by-name terminates
  • Does not contradict confluence, which only says that it is possible to reach the same term, not that a particular evaluation strategy will reach it
  • Recall that primitive recursion trivially guarantees termination
Abstract Data Types

• Consider an abstract data type
  • With $N$ constructors
  • The $i$th constructor has arity $K^i$

• There is a general scheme for encoding such data types where the $i$th constructor has arity $K^i + N$
Example: Lists

Consider the list data type:

List(A):

  nil: List(A)
  cons: A -> List(A) -> List(A)

nil: \lambda n.\lambda c.n
cons: \lambda h.\lambda t.\lambda n.\lambda c. h (t n c)
Equivalences

• The following are all equivalent in computational power
  • SKI calculus
  • Lambda calculus
  • Turing machines

• Next time we will talk about typed lambda calculus, which is strictly less powerful.