Lambda Calculus

Kenneth Lai

Brandeis University

September 7, 2022

Announcements

- By 11:59pm today
  - Personal Learning Goals Part 1 due
- For next Monday
  - If you can, do exercises 7, 8, 9, and 11 (on last slide)
- For 9/21
  - HW1 due

Today's Plan

- More Tools for Formal Semantics
  - Lambda Calculus
  - Types
  - Functional Programming with Haskell

Today's Plan

- More Tools for Formal Semantics
  - Lambda Calculus
  - Types
  - Functional Programming with Haskell
  - (we'll see how far we get...)

Computational Semantics
Day 3: Lambda calculus and the composition of meanings

Jan van Eijck[1] & Christina Unger[2]

[1]CWI, Amsterdam, and UiL-OTS, Utrecht, The Netherlands
[2]CITEC, Bielefeld University, Germany

ESSLLI 2011, Ljubljana

Lambda calculus

Lambdas changed my life.

(Barbara H. Partee)

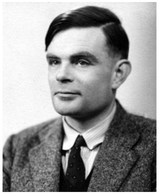
All you need is lambda.

(Simon Peyton-Jones)

History

In 1936, Turing and Church independently introduced two equivalent models of computation:

- **Alan Turing: Turing Machine**
  A function is computable if a sequence of instructions can be specified and then carried out by a simple abstract computational device.
- **Alonzo Church: Lambda Calculus**
  Every computable function is a function that is definable in the lambda calculus.



Connection to programming languages

- **Imperative programming languages** are based on the way a Turing machine is instructed.
- **Functional programming languages** are based on the lambda calculus.

In fact, the lambda calculus is the smallest universal programming language of the world (universal, because any computable function can be expressed and evaluated).

- Expressions correspond to programs.
- The reduction of an expression corresponds to program execution.

In fact, it is the core of functional programming languages, which are basically executable (typed) lambda calculi extended with constants, datatypes, input/output, etc.

Lambda calculus

The lambda calculus is a formal system for defining and investigating functions.

Two basic concept:

- function abstraction for representing functions, using a variable-binding operator λ
- function application, corresponding to substitution of bound variables

Formal definition and properties

Lambda calculus: Formal definition

Variables v and expressions E are defined as follows:

$$v ::= x \mid v'$$
$$E ::= v \mid \lambda v.E \mid (E\ E)$$

Variables

$$v ::= x \mid v'$$
$$E ::= v \mid \lambda v.E \mid (E\ E)$$

For our purposes, we write **variables** as lower case letters $x, y, z, \ldots$, possibly with indices.

Haskell: Variables (including function names) begin with a lower case letter.

- `x`, `x'`, `x1`
- `variable`, `newVAR`, `my_variable`
- ...

Function abstraction

$$v ::= x \mid v'$$
$$E ::= v \mid \lambda v.E \mid (E\ E)$$

$\lambda v.E$ represents a function, where v is the variable abstracted over (bound by the operator λ), and E is the body of the function.

Examples: $\lambda x.x$, $\lambda x.\lambda y.x$

Haskell: Function abstraction is written as `\ v -> E`.

- `\ x -> x`
- `\ x -> (\ y -> x)`
  or shorter: `\ x y -> x`

Function application

$$v ::= x \mid v'$$
$$E ::= v \mid \lambda v.E \mid (E\ \ E)$$

Function application represents applying an expression to another expression, e.g. a function to an argument.

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

Haskell: Function application is written as E E.

- `(\ x -> x) y`
- `(\ x y -> x) z`

Function application

Note: Function application can also be written as $(E)(E)$, with parentheses or square brackets for readability

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

Reducing expressions

Function application expressions can be reduced to simpler expressions. This corresponds to substitution of bound variables.

Reduction rule (called *beta reduction*):

$$(\lambda v.E_1 \quad E_2) \ \rhd \ E_1 \, [v := E_2]$$

Where $E_1 \, [v := E_2]$ denotes the substitution of E_2 for all free occurrences of v in E_1.

Example:

- $(\lambda x.(x \ y) \quad \lambda z.z) \ \rhd$

Reducing expressions

Function application expressions can be reduced to simpler expressions. This corresponds to substitution of bound variables.

Reduction rule (called *beta reduction*):

$$(\lambda v.E_1 \quad E_2) \;\rhd\; E_1\,[v := E_2]$$

Where $E_1\,[v := E_2]$ denotes the substitution of E_2 for all free occurrences of v in E_1.

Example:

- $(\lambda x.(x\ \ y)\ \ \lambda z.z) \;\rhd\; (\lambda z.z\ \ y) \;\rhd$

Reducing expressions

Function application expressions can be reduced to simpler expressions. This corresponds to substitution of bound variables.

Reduction rule (called *beta reduction*):

$$(\lambda v.E_1 \quad E_2) \ \triangleright \ E_1 \, [v := E_2]$$

Where $E_1 \, [v := E_2]$ denotes the substitution of E_2 for all free occurrences of v in E_1.

Example:

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

Free and bound variables

An occurrence of the variable v in the expression E is bound if it is in the scope of a lambda prefix λv.

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

Free and bound variables

An occurrence of the variable v in the expression E is bound if it is in the scope of a lambda prefix λv.

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

Note: When substituting expressions, we have to make sure that no variables get accidentally captured.

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

This can be ensured by variable renaming.

Free and bound variables

Note: Other prefixes such as $\forall v$ and $\exists v$ (which we will learn more about in a week or so) also bind occurrences of v within their scopes

Free and bound variables

Note: Other prefixes such as $\forall v$ and $\exists v$ (which we will learn more about in a week or so) also bind occurrences of v within their scopes

- Variable renaming is also called alpha conversion

Observation

Reductions need not come to an end.

- $(\lambda x.(x\ x)\ \ \lambda x.(x\ x))$
- $(\lambda x.((x\ x)\ x)\ \ \lambda x.((x\ x)\ x))$

Confluence

The result of beta reduction is independent from the order of reduction, i.e. if an expression can be evaluated in two different ways and both ways terminate, then both ways will yield the same result (*Church-Rosser theorem*).

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

Note: The reduction order does, however, play a role for efficiency and can influence whether a reduction terminates or not.

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

Conventions

- Applications associate to the left; thus, when applying a function to a number of arguments, we can write $f\,x\,y\,z$ instead of $(((f\ x)\ y)\ z)$.
- The body of a lambda abstraction (the part after the dot) extends as far to the right as possible. I.e., $\lambda x.E_1\ E_2$ means $\lambda x.(E_1\ E_2)$, and not $(\lambda x.E_1)\ E_2$.

Adding function constants

Lambda calculus as we saw it is already enough to define natural numbers and arithmetic operations. We can abbreviate the corresponding expressions by adding constants to the language:

- 1,2,3. . . for natural numbers
- $+$ and * for addition and multiplication

Analogously, we can add constants a, b, c for entities, *wizard* for unary functions, *admire* for binary functions, and so on.

Exercises

- Exercises from Coppock and Champollion (2022) Chapter 5, Exercise 3
- For each of the following lambda expressions, apply beta reduction to give a completely reduced expression (i.e., in beta normal form):
  1. $[\lambda x.P(x)](a)$
  4. $[\lambda x.R(y,a)](b)$

Exercises

- Exercises from Coppock and Champollion (2022) Chapter 5, Exercise 3
- For each of the following lambda expressions, apply beta reduction to give a completely reduced expression (i.e., in beta normal form):
  7. $[[\lambda x \lambda y . R(x, y)](b)](a)$
  8. $[\lambda x . [\lambda y . R(x, y)](b)](a)$
  9. $[\lambda X . \exists x . [P(x) \wedge X(x)]](\lambda y . R(a, y))$
  11. $[\lambda X . \exists x . [P(x) \wedge X(x)]](\lambda y . R(y, x))$