let x = 1 in ...

Programming Paradigms and Formal Semantics

The Untyped Lambda Calculus

Ralf Lämmel

Resources: The slides of this lecture were derived from [Järvi], with permission of the original author, by copy & paste or by selection, annotation, or rewording. [Järvi] is in turn based on [Pierce] as the underlying textbook.

[Järvi] Slides by J. Järvi: “Programming Languages”, CPSC 604 @ TAMU (2009)
What’s the lambda calculus?

• It is the core of functional languages.
  • It is a mathematical system for studying programming languages.
    ✦ design, specification, implementation, type systems, et al.
  • It comes in variations of typing: implicit/explicit/none.
• Formal systems built on top of simply typed lambda calculus:
  ✦ System F — for studying polymorphism
  ✦ System F<: — for studying subtyping
  ✦ ...
Lambda calculus — examples

\[ \lambda x. x \]
\[ \lambda f. \lambda g. \lambda x. f(g \, x) \]

The identity function

Function composition

Examples

Some lambda terms

Compare lambda and, say, Haskell constructs:
Lambda calculus — overview

• Abstract syntax:

\[
M ::= x \mid M \ M \mid \lambda x. M
\]

• \(M\) is a lambda term.

• An infinite set of variables \(x, y, z, \ldots\) is assumed.

• \(M \ N\) is an application.

  Function \(M\) is applied to the argument \(N\).

• \(\lambda x. M\) is an abstraction.

  The resulting function maps \(x\) to \(M\).

Lambda functions are anonymous.
What’s the lambda calculus?

• Church’s thesis:

All intuitively computable functions are \( \lambda \)-definable.

• An established equivalence of notions of computability:

Set of Lambda-definable functions

= Set of Turing-computable functions
Formalization of the lambda calculus

• Syntax

\[ t ::= x \]

\[ \lambda x.t \quad \text{Terms} \]

\[ t \ t \]

\[ v ::= \lambda x.t \quad \text{Values (normal forms)} \]

• Evaluation

\[ \frac{t_1 \to t_1'}{\frac{t_2 \to t_1'}{t_1 \ t_2 \to t_1'} \ t_2} \]

\[ \frac{t \to t'}{\frac{v \ t \to v \ t'}{v \ t \to v \ t'}} \]

\[ (\lambda x.t) \ v \to [v/x]t \]

Reduce function position, then reduce argument position, then apply.
Syntactic sugar and conventions

- $M \, N_1 \, \ldots \, N_k$ means $\ldots((M \, N_1) \, N_2) \, \ldots \, N_k$.

  Function application groups from left to right.

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

  Function application has higher precedence.

- $\lambda x_1 x_2 \, \ldots \, x_k. \, M$ means $\lambda x_1.(\lambda x_2.(\ldots(\lambda x_k.(M)) \, \ldots)))$. 
Variable binding

• \( \lambda \) is a binding operator:
  
  It binds a variable in the scope of the lambda abstraction.

• Examples:
  
  ✦ \( \lambda x.M \quad x \text{ is bound (in the lambda abstraction)} \)
  
  ✦ \( \lambda x.x \ y \quad y \text{ is not bound (in the lambda abstraction)} \).

• If a variable occurs in an expression without being bound, then it is called a \textbf{free} occurrence, or a free variable. Other occurrences of variables are called \textbf{bound}.

• A \textbf{closed term} is one without free variable occurrences.
Variable binding — precise definition

$\text{FV}(M)$ defines the set of free variables in the term $M$

$\text{FV}(x) = \{x\}$

$\text{FV}(MN) = \text{FV}(M) \cup \text{FV}(N)$

$\text{FV}(\lambda x. M) = \text{FV}(M) \setminus \{x\}$
Exercise: what are the free and bound variable occurrences in these terms?

\[(\lambda x. y)(\lambda y. y)\]
\[\lambda x. (\lambda y. x\ y)y\]
Substitution and $\beta$-equivalence

• **Computation for the $\lambda$-calculus is based on substitution.**

• Substitution is defined by the equational axiom:

$$(\lambda x. M)N = [N/x]M$$

• Think of substitution as invoking a function:

★ $(\lambda x. M)$ is the function,

★ $N$ is the argument,

★ Substitution takes care of parameter passing.

The terms on both sides are also called $\beta$-equivalent.
\(\alpha\)-equivalence and conversion

• Names of bound variable are insignificant.

\(\lambda x.x\) defines the same function as \(\lambda y.y\)

• Suppose two terms differ only on the names of bound variables.

Then, they are said to be \(\alpha\)-equivalent ( \(\equiv\alpha\) ).

• Equational axiom:

\(\lambda x.M = \lambda y.\left[y/x\right]M\)

where \(y\) does not appear in \(M\)

and substitution applies to free occurrences only.

Performing such renaming is also called \(\alpha\)-conversion.
Reduction $M \rightarrow N$

- **Redex** (reducible expression)
  - $(\lambda x . M )N$
  - Apply substitution; this is called $\beta$-reduction.
    $$(\lambda x . M )N \rightarrow [N/x]M$$

- Computation with the lambda calculus is then a series of
  - $\beta$-reductions, and
  - $\alpha$-conversions.

- **Normal form**:
  - a term that cannot be reduced further.
Inductive definition of substitution

$$[N/x]x = N$$
$$[N/x]y = y, y \text{ any variable different from } x$$
$$[N/x](M_1 M_2) = ([N/x]M_1)([N/x]M_2)$$
$$[N/x](\lambda x. M) = \lambda x. M$$
$$[N/x](\lambda y. M) = \lambda y.([N/x]M), y \text{ not free in } N$$

Examples:

- $$[z/x]x \rightarrow z$$
- $$[z/x](\lambda x.x x) \rightarrow \lambda x.x x$$
- $$[z/x](\lambda y.y x) \rightarrow \lambda y.y z$$
- $$[z/x](\lambda z.x z) \rightarrow \lambda a.z a$$
Properties of reduction (i.e., semantics)

• How do we select redexes for reduction steps?
• Does the result depend on such a choice?
• Does reduction ultimately terminate with a normal form?
Illustration of different reductions
(We assume natural numbers with “+”.)

Option 1

\[
\begin{align*}
\left(\lambda f.\lambda x.f(f\ x)\right)\left(\lambda y.y + 1\right) & \to \left(\lambda x.\left(\lambda y.y + 1\right) (\lambda y.y + 1\ x)\right) \\
& \to \left(\lambda x.\left(\lambda y.y + 1\right) (x + 1)\right) \\
& \to \left(\lambda x.(x+1+1)\right) \\
& \to (2+1+1) \\
& \to 4
\end{align*}
\]

Option 2

\[
\begin{align*}
\left(\lambda f.\lambda x.f(f\ x)\right)\left(\lambda y.y + 1\right) & \to \left(\lambda x.\left(\lambda y.y + 1\right) (\lambda y.y + 1\ x)\right) \\
& \to \left(\lambda y.y + 1\right) (\lambda y.y + 1\ x) \\
& \to (\lambda y.y + 1\ 2) \\
& \to ... \\
& \to ... \\
& \to 4
\end{align*}
\]
Confluence

• Confluence: evaluation strategy is not significant for final value.

• That is: there is (at most) one normal form of a given expression.

Strong normalization property of a calculus with reduction

• Definition:

For every term $M$ there is a normal form $N$ such that $M \rightarrow^* N$.

• Strong normalization properties for lambda calculi:

♦ Untyped lambda calculus: *no* $(\lambda x.x\ x)(\lambda x.x\ x)$

♦ Simply-typed lambda calculus: *yes*
Evaluation/reduction strategies

✦ Full beta reduction

Reduce anywhere.

✦ Applicative order ("reduce the leftmost innermost redex")

Reduce argument before applying function.

✦ Normal order ("reduce the leftmost outermost redex")

Apply function before reducing argument.

Choice of strategy may impact termination behavior

- eager (strict)
- lazy (non-strict)

Extension vs. encoding

• Typical extensions
  (giving rise to so-called applied lambda calculi)
  ✦ Primitive types (numbers, Booleans, ...)
  ✦ Type constructors (tuples, records, ...)
  ✦ **Recursive functions**
  ✦ Effects (cell, exceptions, ...)
  ✦ ...

• Many extensions can be encoded in theory in terms of pure lambda calculus, except that such encoding is somewhat tedious.
Church Booleans

• Encodings of literals
  ✦ true = \lambda t.\lambda f.t
  ✦ false = \lambda t.\lambda f.f

• Conditional expression (if)
  ✦ Expectations
    ★ test b v w \rightarrow^* v, if b = true
    ★ test b v w \rightarrow^* w, if b = false
  ✦ Encoding
    ★ test = \lambda l.\lambda m.\lambda n.l m n
Example reduction

\[(\lambda l.\lambda m.\lambda n.l\ m\ n)\ \text{true} \ v\ w\]

\[\rightarrow (\lambda m.\lambda n.\text{true}\ m\ n)\ v\ w\]

\[\rightarrow (\lambda n.\text{true}\ v\ n)\ w\]

\[\rightarrow \text{true}\ v\ w\]

\[\rightarrow (\lambda t.\lambda f.t)\ v\ w\]

\[\rightarrow (\lambda f.v)w\]

\[\rightarrow v\]

Likewise for numerals or pairs.
Church numerals

• Encodings of numbers
  ✦ \( c_0 = \lambda s.\lambda z.z \)
  ✦ \( c_1 = \lambda s.\lambda z.s \, z \)
  ✦ \( c_2 = \lambda s.\lambda z.s \, (s \, z) \)
  ✦ \( c_3 = \lambda s.\lambda z.s \, (s \, (s \, z)) \)
  ✦ ...

• Encodings of functions on numbers
  ✦ \( \text{succ} = \lambda n.\lambda s.\lambda z.s \, (n \, s \, z) \)
  ✦ \( \text{plus} = \lambda m.\lambda n.\lambda s.\lambda z.m \, s \, (n \, s \, z) \)
  ✦ \( \text{times} = \lambda m.\lambda n.m \, (\text{plus} \, n) \, c_0 \)
  ✦ ...

A numeral \( n \) is a lambda abstraction that is parameterized by a case for zero, and a case for succ. In the body, the latter is applied \( n \) times to the former. This caters for primitive recursion (akin to the Boolean TEST).
Recursive functions

• Let us define the factorial function.

• Suppose we had “recursive function definitions”.

\[ f \equiv \lambda n. \text{if } n == 0 \text{ then } 1 \text{ else } n \times f(n - 1) \]

• Can we do such recursion with anonymous functions.

• We can use a fixed point combinator instead.
Fixed points

• Consider a function $f : X \to X$.
• A fixed point of $f$ is a value $z$ such that $z = f(z)$.
• A function may have none, one, or multiple fixed points.
• Examples (functions and sets of fixed points):
  ✦ $f(x) = 2x$ \{0\}
  ✦ $f(x) = x$ \{0, 1,...\}
  ✦ $f(x) = x + 1$ \\$
• We call $Y$ a fixed-point combinator if it satisfies the following \textbf{definitional property}:
  \begin{align*}
  \text{For all } f : X \to X \text{ it holds that } Y f &= f (Y f)
  \end{align*}

100$ Question: does such a $Y$ exist?
Defining factorial as a fixed point

- Start from a recursive definition.
  \[ f \equiv \lambda n. \text{if } n == 0 \text{ then } 1 \text{ else } n*f(n - 1) \]

- Eliminate self-reference; receive function as argument.
  \[ g \equiv \lambda h.\lambda n.\text{if } n == 0 \text{ then } 1 \text{ else } n*h(n - 1) \]
  \[ \star g \text{ takes a function } (h) \text{ and returns a function.} \]

- Define f as a fixed point.
  \[ f \equiv Y \ g \]
Fixed points cont’d

- For example, apply definitional property to factorial $g$:

\[(Y g) \ 2\]

\[= [Y \text{ def. prop.}] \quad g \ (Y g) \ 2\]

\[= [\text{unfold } g] \quad (\lambda \ h. \lambda \ n. \ \text{if } \ n == 0 \ \text{then} \ 1 \ \text{else} \ n*h(n-1)) \ (Y g) \ 2\]

\[= [\text{beta reduce}] \quad (\lambda \ n. \ \text{if } \ n == 0 \ \text{then} \ 1 \ \text{else} \ n*((Y g)(n-1))) \ 2\]

\[= [\text{beta reduce}] \quad \text{if } \ 2 == 0 \ \text{then} \ 1 \ \text{else} \ 2*((Y g)(2-1))\]

\[= [\text{"-" reduce}] \quad \text{if } \ 2 == 0 \ \text{then} \ 1 \ \text{else} \ 2*((Y g)(1))\]

\[= [\text{"if" reduce}] \quad 2*((Y g)(1))\]

\[= \ldots\]

\[= 2\]

This is as if we had extended evaluation.

Apply these steps one more time.
A lambda term for $Y$

• One option:
  $$Y = \lambda f . (\lambda x. f (x x)) (\lambda x. f (x x))$$

• Verification of the definitional property:
  $$Y \ g = g \ (Y \ g)$$

• Proof:

  $$Y \ g$$

  $$= [\text{unfold } Y] \ (\lambda f . (\lambda x. f (x x)) (\lambda x. f (x x))) \ g$$

  $$= [\text{beta reduce}] \ (\lambda x. g(x x)) (\lambda x. g(x x)))$$

  $$= [\text{beta reduce}] \ g \ ((\lambda x. g(x x)) (\lambda x. g(x x)))$$

  $$= [\text{fold } Y] \ g \ (Y \ g)$$

Not suitable for applicative order.
• **Summary:** The untyped lambda calculus
  - A concise core of functional programming.
  - A foundation of computability.

• **Prepping:** “*Types and Programming Languages*”
  - Chapter 5

• **Lab:** NB and the lambda calculus in Prolog

• **Outlook:**
  - The simply-typed lambda calculus