The λ-calculus

© Vivian McPhail

1 September 2015

Outline

Notation (Meta-variables)

An infinite supply of variables, V, are denoted by lower-case roman letters (x, y, z, $\ldots$) and λ-terms, Λ, are denoted by upper-case roman letters (M, N, O, $\ldots$).

Definition (Lambda Terms)

A λ-term is either a variable, an abstraction, or an application:

$$\begin{array}{llll} x \in V & \Rightarrow & x & \in \Lambda \quad \textit{Variable} \\ x \in V, M \in \Lambda & \Rightarrow & (\lambda x.M) & \in \Lambda \quad \textit{Abstraction} \\ M, N \in \Lambda & \Rightarrow & (MN) & \in \Lambda \quad \textit{Application} \end{array}$$

Notation (Bracketing)

1. $(M(NO)) \rightarrow M(NO)$ *Outermost brackets may be discarded.*
2. $(MN)O \rightarrow MNO$ *Application associates to the left.*
3. $\lambda x.(\lambda y.(\lambda z.M)) \rightarrow \lambda x.\lambda y.\lambda z.M$ *Abstraction associates to the right.*
4. $\lambda x.\lambda y.\lambda z.M \rightarrow \lambda x\, y\, z.M$ *Consecutive abstractors can be abbreviated as one.*

Definition (Bound Variables)

In a λ-term, $\lambda x.M$, the variable x is bound by the abstractor λ and is under the abstractor's scope.

Definition (Free Variables)

1. *A variable occuring in a λ-term that is not bound is free. The set of free variables is defined inductively:*

$$
\begin{array}{rcl}
FV(x) & = & \{x\} \\
FV(\lambda x.M) & = & FV(M) \setminus \{x\} \\
FV(MN) & = & FV(M) \cup FV(N)
\end{array}
$$

2. *M is closed or a combinator if $FV(M) = \{\emptyset\}$.*
3. $\Lambda^0 = \{M \in \Lambda \mid M \textit{is closed}\}$.

Notation (Syntactic Equality)

For any two λ-terms M and N, $M \equiv N$ denotes syntactic equality.

Definition (α-conversion, Church 1941)

The renaming of bound variables is called α-conversion.

$$\lambda x.M \xrightarrow{\alpha} \lambda y.[y/x]M \quad \text{provided that } y \text{ does not occur in } M$$

Two terms that are the same up to renaming of variables are equivalent.

$$M \xrightarrow{\alpha} N \Rightarrow M \equiv N$$

De Bruijn terms, which label variables by position (distance from the closest left-hand abstractor) are a means of avoiding variable renaming issues Barendregt (1984).

Outline

Definition (Substitution)

The expression $[N/x]M$ is the term M with all free occurrences of x substituted for by N:

$$
\begin{array}{rcl}
[N/x]x & \rightarrow & N \\
[N/x]y & \rightarrow & y \\
[N/x](\lambda y.M) & \rightarrow & \lambda y.([N/x]M) \\
[N/x](M_1 M_2) & \rightarrow & ([N/x]M_1)([N/x]M_2)
\end{array}
$$

Outline

Definition (μ-conversion)

λ-terms can be given names and the μ-conversion of a term involving these names is the term with the name replaced by the corresponding λ-term

$$\textbf{name} := M \Rightarrow (\textbf{name}\ N) \xrightarrow{\mu} MN$$

Definition (Standard Combinators)

$$\begin{array}{rcl} \mathbf{I} & := & \lambda x.x \\ \mathbf{K} & := & \lambda x\,y.x \\ \mathbf{S} & := & \lambda x\,y\,z.x\,z\,(y\,z) \end{array}$$

Theorem

$\forall M \in \Lambda, \mathbf{SK}M = \mathbf{I}$

Outline

Definition (β-reduction)

The principal axiom scheme is β-reduction:

$$(\lambda x.M)N \xrightarrow{\beta} [N/x]M \quad \forall M, N \in \Lambda$$

Theorem (FixedPoint Theorem)

1. $\forall F \in \Lambda, \exists X \in \Lambda : FX = X$
2. *There is a fixed point combinator*

$$\mathbf{Y} := \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$$

such that

$$\forall F \in \Lambda, F(\mathbf{Y}F) = \mathbf{Y}F$$

Definition (η-Reduction)

Redundant abstractions can be removed with η-reduction

$$\lambda x.Mx \xrightarrow{\eta} M \quad x \notin FV(M)$$

The theory λ extended with η-reduction is called λ_η.

Definition (Redex)

A redex is a reducible expression, which is an expression to which a reduction can be applied.

Example (Reducible Expressions)

Reducible expressions:

1. *A variable, a, is* **NOT** *a redex.*
2. *The abstraction* $\lambda x.My$ *is a redex.*
3. *The application* MN *is a redex if* M *is an abstraction.*

Outline

Definition (β-Normal Form)

Let $M \in \Lambda$.

1. *M is a **β-normal form** (β-nf or nf) if M has no subterm $(\lambda x.P)Q$.*
2. *M has a β-normal form if $\exists N : N = M$ and N is a β-normal form.*
3. *If M is a nf, it is also said that M is* **in** *nf.*

Definition ($\beta\eta$-Normal Form)

1. *M is a $\beta\eta$-normal form if M has no subterm $(\lambda x.P)Q$ or $(\lambda x.R\,x)$ with $x \notin FV(R)$.*
2. *M has a $\beta\eta$-normal form if*

$$\exists N : \lambda_\eta \vdash M = N \wedge N \textit{ is a } \beta\eta\textit{-normal form}$$

Definition (Head and Arguments)

In a λ-term $\lambda\vec{x}.M\vec{N}$ where the x_i may occur in M, M is the head and the $\vec{N}$ are the arguments.

Definition (Head Normal Form)

1. *M is a head normal form (hnf) if M has the form $M \equiv \lambda\vec{x}.y\,\vec{N}$.*
2. *M has a hnf if $\exists N : M = N$ and N is a hnf.*

Definition (Weak Head Normal Form)

1. *M is a weak head normal form (whnf) if M has the form $M \equiv \lambda\vec{x}.M\vec{N}$ where in the λ-term all reductions to the function (the head, M) have been applied, but not all reductions to the parameter, N, have been applied. That is, M is a whnf if it is a hnf or a lambda abstraction.*
2. *M has a whnf if $\exists N : M = N$ and N is a whnf.*

Outline

Definition (Redexes)

1. *The leftmost redex is the redex whose abstraction is to the left of all other redexes.*
2. *The rightmost redex is the redex whose abstraction is to the right of all other redexes.*
3. *The innermost redex is one that contains no other redexes.*
4. *The outermost redex is one that is contained within no other redexes.*

Definition (Evaluation Order)

1. *Applicative Order evaluates the leftmost, innermost redex.*
2. *Normal Order evaluates the leftmost, outermost redex.*

Definition (Parameter Evaluation)

1. *Call by value evaluates arguments before evaluating the head.*
2. *Call by name evaluates the head before evaluating arguments.*

Definition (Evaluation Strategy)

1. *In strict evaluation, the arguments are fully evaluated before the function is applied.*
2. *In non-strict (or lazy) evaluation, arguments to a function are not evaluated until they are used.*

Definition (Church-Rosser)

1. *Let* $\mathbf{R}$ *be a binary relation on* Λ*. Then* $\mathbf{R}$ *satisfies the diamond property* ($\mathbf{R} \models \diamond$) *if*

$$\forall M, M_1, M_2, (M, M_1) \in \mathbf{R} \wedge (M, M_2) \in \mathbf{R} \\ \Rightarrow \exists M_3 : (M_1, M_3) \in \mathbf{R} \wedge (M_2, M_3) \in \mathbf{R}$$

2. *A notion of reduction* $\mathbf{R}$ *is said to be confluent if* $\overset{\mathbf{R}}{\twoheadrightarrow}$ *satisfies the diamond property.*
3. *A confluent notion of reduction* $\mathbf{R}$ *has the Church-Rosser property.*

Theorem (Church-Rosser)

$\beta\eta$-reduction is confluent and so the λ_η theory is Church-Rosser.

Outline

Definition (Referential Transparency)

A function is referentially transparent when given the same arguments, or their reductions, it returns the same result.

Outline

Definition (Church Numerals)

The Church numerals $\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_n$ *are defined by*

$$\mathbf{c}_n := \lambda f\, x. f^n(x)$$

Lemma

1. $(\mathbf{c}_n x)^m(y) = x^{n*m}(y)$
2. $(\mathbf{c}_n)^m(x) = \mathbf{c}_{(n^m)}, \quad m > 0$

Proposition (J. B. Rosser)

Define

$$\begin{array}{rcl} \mathbf{A}_+ & := & \lambda x\, y\, p\, q. x\, p (y\, p\, q) \\ \mathbf{A}_* & := & \lambda x\, y\, z. x (y\, z) \\ \mathbf{A}_{exp} & := & \lambda x\, y. y\, x \end{array}$$

Then $\forall m, n \in \mathbb{N}$

1. $\mathbf{A}_+ \mathbf{c}_m \mathbf{c}_n = \mathbf{c}_{m+n}$
2. $\mathbf{A}_* \mathbf{c}_m \mathbf{c}_n = \mathbf{c}_{m*n}$
3. $\mathbf{A}_{exp} \mathbf{c}_m \mathbf{c}_n = \mathbf{c}_{(m^n)}, \;\; m \neq 0$

Outline

```
S = (^x y z.x z(y z))
K = (^x y.x)
I = (S K K)

false = (S K)
true = K

zero = false
succ = (^n f x.f (n f x))
one = (succ zero)

if = I
isZero = (^n.n (^z.false) true)

mul = (^m n f.m (n f))
pred = (^n f x.n (^g h.h (g f)) (^u.x) (^u.u))
```

```
Y = (^x.(^y.x(y y))(^y.x(y y)))

nextFact = (^f n.if (isZero n) one (mul n (f (pred n))))
fact = (Y nextFact)
```

A lambda interpreter:

```
$ git clone https://github.com/amcphail/lambda
$ ghci

GHCi, version 7.10.2: http://www.haskell.org/ghc/  :? for help
Prelude> :l Lambda
[1 of 3] Compiling Parsers          ( Parsers.hs, interpreted )
[2 of 3] Compiling Combinators      ( Combinators.hs, interpreted )
[3 of 3] Compiling Lambda           ( Lambda.hs, interpreted )
Ok, modules loaded: Lambda, Parsers, Combinators.

*Lambda> let fact n = if n == 0 then 1 else (n * (fact (n-1)))
*Lambda> fact 4
24

*Lambda> main

lambda term? fact four
-> ^f x.f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f x\
)))))))))))))))))))))))

lambda term?
```

Barendregt, H. (1984). *The Lambda Calculus: Its Syntax and Semantics*, Vol. 103 of *Studies in Logic*, second, revised edn, North-Holland, Amsterdam.

Barendregt, H. and Barendsen, E. (1994). Introduction to lambda calculus. http://citeseer.ist.psu.edu/barendregt94introduction.html.