AN OVERVIEW OF THE LAMBDA CALCULUS

THOMAS HERZOG

Abstract. This paper gives an overview of the λ-calculus. It introduces the basic notation and theory, develops important results and emphasizes the importance of the λ-calculus, being the foundation of computation in general.

Date: September 30, 2007.

Contents

I can't imagine a universe without the logics that we know. And therefore I cannot imagine a universe without lambda calculus.

Philip Wadler [8]

1. Introduction

One of the fundamental issues of computer science is computability: The question of which kinds of functions can be computed mechanically, i.e. by a computer. Some attempts to formalize this problem have been developed in the first half of the last century, including *combinatory logic* by Haskell Curry and Moses Schönfinkel, and, a bit later, the λ-*calculus* by Alonzo Church ([3]). This paper will give a succinct survey of the λ-calculus, serving as an introduction to the theory and its implications. Important results will be listed and selectively proven. For a more thorough formal treatment, the reader is refered to [5], which serves as an excellent primer to the λ-calculus, and uses mostly the same notation as we will employ here.

2. The Basics

In this section we will introduce the basic notation and theory for the λ-calculus.

2.1. **Notation.** The alphabet for terms in the *lambda*-calculus consists of:

(1) Variables: $v_1, v_2, v_3, \ldots$
(2) The lambda symbol: λ
(3) Parentheses: (,)

Since one could construct arbitrary 'terms' from this alphabet, we now have to define which terms are actually *well-formed*:

Definition 1. λ-term
We define the class Λ of λ-terms as the least class with the following properties:

(1) $x \in \Lambda$, x being a variable
(2) $M \in \Lambda \quad \Longrightarrow \quad (\lambda x M) \in \Lambda$ (abstraction)
(3) $M, N \in \Lambda \quad \Longrightarrow \quad (MN) \in \Lambda$ (application)

Thus, for example, $x, (\lambda yz), (xy)$ and $(((\lambda xx)y)z)$ are all terms. (Note: In the following, we will write *term* to mean *well-formed term.*)

It is obvious that with more complex formulae, one quickly loses sight of the formula itself amidst a jungle of parentheses. Hence we introduce two abbreviated notations that are widely employed in practise:
(Note: $\equiv$ denotes syntactic equality.)

$$\lambda x_1 x_2 \ldots x_n . M \equiv (\lambda x_1 (\lambda x_2 (\ldots (\lambda x_n M) \ldots)))$$

and

$$M N_1 N_2 \ldots N_n \equiv (\ldots (M N_1) \ldots N_n)$$

For example, $(\lambda x(\lambda yx))$ becomes $\lambda xy.x$.

In the following section we will also need the notion of subterms:

Definition 2. λ-subterm
We define the function $Sub : \Lambda \to \mathfrak{P}(\Lambda)$ for subterms inductively:

$$\begin{aligned} Sub\ x &= \{x\} \\ Sub(\lambda x M) &= (Sub\ M) \cup \{(\lambda x M)\} \\ Sub(MN) &= (Sub\ M) \cup (Sub\ N) \cup \{(MN)\} \end{aligned}$$

2.2. **Free and Bound Variables.** We now come to the important concept of free and bound variables. Consider the following example:

$$(\lambda xy.x)xz$$

Is x bound or free? The answer is: both, depending on which scope we're looking at. The x inside the subterm denoted by the parentheses is bound by the λ preceding it, but the x outside the parentheses is not bound by any λ, so it is free. We will formalize this as follows:

Definition 3. Bound Variables
The set of bound variables for a λ-term is given by the function $BV : \Lambda \to \mathfrak{P}(\Lambda)$, defined inductively as:

$$\begin{aligned} BV\ x &= \varnothing \\ BV(\lambda x M) &= (BV\ M) \cup \{x\} \\ BV(MN) &= (BV\ M) \cup (BV\ N) \end{aligned}$$

Definition 4. Free Variables
The set of free variables for a λ-term is given by the function $FV : \Lambda \to \mathfrak{P}(\Lambda)$, defined inductively as:

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

This notion of variable binding should be familiar to anyone used to the $\exists$ and $\forall$ quantifiers of predicate calculus.

Definition 5. Closed Term
The class of *closed λ-terms*, denoted Λ^0, is the class of λ-terms containing no free variables:

$$\Lambda^0 := \{M \in \Lambda : FV\ M = \varnothing\}$$

Closed terms are also often called *combinators*.

Combinators intuitively represent functions in programming languages. For example, consider the following Python[1] code:

```
def id(x):
  return x
```

This (identity) function can be expressed in the λ-calculus as $\lambda x.x$, with x being the function parameter for the (nameless) function, and x also being the 'result'.

[1]We use Python instead of Java, because Java is strongly typed and would require an entire class of similar functions for each different type of the parameter x.

But what about more than one parameter? This is easily accomplished by serial application of two abstractions, for example:

```
def swap(x,y):
  return (y,x)
```

Here we take a parameter tuple and return the mirrored tuple. This corresponds to the λ-term $\lambda xy.yx \equiv (\lambda x(\lambda y.yx))$. This technique is called *currying*, in honour of H. Curry.

2.3. **The theory λ.** We now know how to construct terms, but they are static. We can't *do* anything with them yet. For the developed notation to be of any practical value, we need to introduce rules of inference, so we can derive new theorems from axioms and previously known theorems. We also want to be able to evaluate expressions: to apply terms to a given abstraction and compute the result.

$(\lambda x.M)N = M[x := N]$	(β)
$M = M$	reflexivity of convertibility
$\dfrac{M = N}{N = M}$	commutativity of convertibility
$\dfrac{M = N \quad N = L}{M = L}$	transitivity of convertibility
$\dfrac{M = N}{MZ = NZ}$	
$\dfrac{M = N}{ZM = ZN}$	
$\dfrac{M = N}{\lambda x.M = \lambda x.N}$	weak extensionality (ζ)

Table 1. The theory λ

Table 1 summarizes the theory λ. Observe that rule (β) corresponds to function application, i.e. $\lambda x.M$ is a function with parameter x and function body M in which we *substitute* all occurences of x with N.

This theory, along with the previously introduced notation, is called the λ-calculus. If $M = N$ is a theorem of λ, we say M and N are *convertible* and write

$$\lambda \vdash M = N$$

It follows that:

$$M \equiv N \implies M = N$$

but not the opposite:

$$\neg(M = N \implies M \equiv N)$$

2.4. **Substitution.** We have to handle substitution carefully, though. Consider the following example, in which we employ a naive (and wrong) approach to substitution:

$$(\lambda xy.yx)y \neq \lambda y.yy$$

Here the argument y, which should have stayed a free variable after application, has become bound to the second λ abstraction, only because it was unlucky enough to share the name of the abstraction's formal parameter y. This problem is called *variable capture*. To avoid it, we will employ the *variable convention*, introduced in [1].

Before we can give a definition, we need to introduce some other concepts.

Definition 6. Contexts

The class of λ-contexts $\mathcal{C}[]$ is the least class with the following properties:

(1) $x \in \mathcal{C}[]$
(2) $[] \in \mathcal{C}[]$
(3) $C_1[], C_2[] \in \mathcal{C}[] \implies (C_1[]C_2[]), (\lambda x C_1[]) \in \mathcal{C}[]$

For example, $C[] \equiv \lambda x.[]x$ is a context (written in the usual abbreviated style introduced earlier). If we want to fill the 'hole' with $\lambda y.y$, we get:

$$C[\lambda y.y] \equiv \lambda x.(\lambda y.y)x$$

Definition 7. Change of Bound Variables

M' is produced from M by a *change of bound variables* if $M \equiv C[\lambda x.N]$ and $M' \equiv C[\lambda y.(N[x := y])]$ where $y \notin BVN \cup FVN$ and $C[]$ is a context with one hole.

Definition 8. α-congruence

M is α-congruent to N, $M \equiv_\alpha N$, if N results from M by a series of changes of bound variable.

For example:

$$\lambda xy.yx \equiv_\alpha \lambda ab.ba \equiv_\alpha \lambda yx.xy$$

But compare to our example of naive substitution:

$$(\lambda xy.yx)y \equiv_\alpha (\lambda xz.zx)y = \lambda z.zy \neq \lambda y.yy$$

Using the notion of α-equivalency, we can now define a safe method of substitution:

Definition 9. Variable Convention

If $M_1, M_2, \ldots, M_n$ occur in a certain context, then in these terms all bound variables are chosen to be different from free variables.

This means that if a substitution $M[x := N]$ threatens to produce a variable capture problem, that is, a free variable that occurs in N appears in M, we replace N with $N' \in [N]_\alpha$ (the equivalence class of N under $\equiv_\alpha$), with the offending variables renamed.

To formalize and summarize our treatment of substitution:

(1) $x[x := N] \equiv N$
(2) $y[x := N] \equiv y$, if $x \neq y$
(3) $(\lambda y.M)[x := M] \equiv \lambda y.(M[x := N])$
(4) $(M_1M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N])$

It is also possible to reorder substitutions, and to formulate a number of useful properties:

Lemma 1. *The Substitution Lemma*

If $x \neq y$ and $x \notin FVL$ then

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

Lemma 2. *Useful substitution properties*

$$\begin{aligned} M = M' &\implies M[x := N] = M'[x := N] \\ N = N' &\implies M[x := N] = M[x := N'] \\ M = M', N = N' &\implies M[x := N] = M'[x := N'] \end{aligned}$$

To get a better feeling for applying λ rules, we will quickly prove property one:

Proof.

$$\begin{aligned} M = M' &\implies \lambda x.M = \lambda x.M' && (\zeta) \\ &\implies (\lambda x.M)N = (\lambda, .M')N \\ &\implies M[x := N] = M'[x := N] && (\beta) \text{ twice} \end{aligned}$$

□

Another very useful and moral[2] result will conclude this section:

Lemma 3. *Leibniz' Law (Referential Transparency)*
Let $C[]$ be a context, then

$$N = N' \implies C[N] = C[N']$$

3. Consistency, Normal Forms and Reduction

Equivalency is a useful concept, but it doesn't help us much if we're interested in treating λ-terms as *programs* and want to *evaluate* them, since there is (as of yet) no clear way of deciding what the 'result' for $M \in \Lambda$ should be.

The solution to this problem is the notion of *reduction*, particularily *β-reduction*, but before we come to that, we have to introduce the concepts of consistency and completeness.

3.1. **Consistency.**

Definition 10. Equations
For $M, N \in \Lambda$,

$$M = N$$

is called an *equation*.
An equation is *closed* **iff** $M, N \in \Lambda^0$

Definition 11. Consistency
Let $\mathcal{T}$ be a theory with equations as formulae. $\mathcal{T}$ is consistent (written $Con(\mathcal{T})$) **iff** $\mathcal{T}$ does not prove every closed equation.
Let $\mathcal{T}$ be a set of equations. $\lambda + \mathcal{T}$ is formed by adding the equations of $\mathcal{T}$ as axioms to λ. Then $Con(\mathcal{T})$ **iff** $Con(\lambda + \mathcal{T})$

Definition 12. Incompatibility
Let $M, N \in \Lambda$. M and N are *incompatible*, written $M \# N$, **iff** $\neg Con(M = N)$.

λ is consistent (see [1] for a proof), but its consistency can be fairly easily disturbed. To demonstrate this, we first introduce three combinators that will turn out to be useful over and over again:

$$\begin{aligned} \mathbf{S} &\equiv \lambda xyz.xz(yz) \\ \mathbf{K} &\equiv \lambda xy.x \\ \mathbf{I} &\equiv \lambda x.x \end{aligned}$$

[2]In the sense of "this *ought* to be true", see [2]

An intuitive understanding of these combinators is that **I** is the identity combinator, **K** projects on the first of two arguments, and **S** takes three arguments, applying the first one to the other two. (Keenan [6] calls **K** the *Kestrel*, **S** the *Starling* and **I** the *Idiot Bird*, because it only chirps back out exactly what it hears. I've often found these analogies helping me to memorize the letters, which is why I mention them here. Also, there's something fiendishly cruel about calling an innocent collection of symbols 'Idiot Bird'. Not to mention funny.)

With these birds in our arsenal (imagine a dull protest from **I**), we can now set out to make λ inconsistent. All it takes is to add the equation:

$$\mathbf{S} = \mathbf{K}$$

The following proof sketch is copied nearly verbatim from [5]:

Proof.

$$\begin{aligned}\mathbf{S} = \mathbf{K} &\implies \mathbf{S}ABC = \mathbf{K}ABC \text{ for all } A, B, C\\ &\implies AC(BC) = AC\end{aligned}$$

In the case $A = C = \mathbf{I}$, and because $\mathbf{I}A = A$ for all A, we get:

$$AC(BC) = AC \implies \mathbf{II}(B\mathbf{I}) = \mathbf{II} \implies B\mathbf{I} = \mathbf{I}$$

If we now set $B = \mathbf{K}D$ for some arbitrary D, we get:

$$B\mathbf{I} = \mathbf{I} \implies \mathbf{K}D\mathbf{I} = \mathbf{I} \implies D = \mathbf{I}$$

Since D was arbitrary, all terms are equal to **I**, meaning $\neg Con(\mathbf{S} = \mathbf{K})$ and thus also $\mathbf{S}\#\mathbf{K}$. □

3.2. **Normal Forms.** Now we introduce the concept of *normal forms*, which can be seen as the 'result' or 'answer' to a given λ-expression. They are the results of reduction, to which we will come in the sequel.

Definition 13. Normal Forms

Let $M \in \Lambda$. M is a *β-normal form*, abbreviated as *β-nf* or *nf* **iff** M has no subterms of the form $(\lambda x.R)S$.

M has a β-nf **iff** $\exists N \in \Lambda : N = M$, with N being a β-nf.

For example, $\lambda x.x$ is a nf. $(\lambda xy.x)(\lambda x.x)$ is not a nf, but has $\lambda xy.y$ as its nf. But consider the following expression:

$$\mathbf{\Omega} \equiv (\lambda x.xx)(\lambda x.xx)$$

This term does not have a nf. (Try applying the (β) rule!)

3.3. **Completeness.** To get to the concept of completeness, we introduce a new axiom:

$$\lambda x.Mx = M, x \notin FVM \quad (\eta)$$

If we add this axiom to λ, we call the resulting theory $\lambda\eta$. $\lambda\eta$ is consistent. Analogous to β-nfs, a $\beta\eta$-nf is a β-nf that does not contain any subterms of the form $(\lambda x.Rx)$ with $x \notin FVR$.

Note that $\lambda x.Mx = M$ is a theorem of $\lambda\eta$, but not of λ.

Proposition 1. *Completeness*

Let $M, N \in \Lambda$ have normal forms. Then either:

$$\lambda\eta \vdash M = N$$

or:

$$\lambda\eta + (M = N) \text{ is inconsistent.}$$

3.4. **Reduction.** We now formally define reductions, that is, relations on λ-terms that allow us to orderly evaluate (reduce) them to normal forms.

Definition 14. Compatibility
Let $R \in \Lambda^2$. R is *compatible* if, for all $M, M' \in \Lambda$ and all contexts $C[]$ with one hole:

$$(M, M') \in R \implies (C[M], C[M'] = \in R$$

Definition 15. Equality
Let $R \in \Lambda^2$. R is an *equality (congruence)* relation if it is a compatible equivalency relation.

Definition 16. Reduction
Let $R \in \Lambda^2$. R is a *reduction* relation if it is compatible, reflexive and transitive.

With these definitions, we will now build up the formalism for β-reduction and β-equality. We will formulate the definitions deliberately general, because they can be applied to other *notions of reduction* R, but the one we will be especially interested is:

$$\beta = \{((\lambda x.M)N, M[x := N]) \ : \ M, N \in \Lambda\}$$

Definition 17. One-step R-reduction
Based on a notion of reduction R, we can construct the compatible closure of R, called a *one-step reduction relation*, as follows:

$$\frac{(M, N) \in R}{M \to_R N}$$

$$\frac{M \to_R N}{MZ \to_R NZ}$$

$$\frac{M \to_R N}{ZM \to_R ZN}$$

$$\frac{M \to_R N}{\lambda x.M \to_R \lambda x.N}$$

Definition 18. R-reduction
The *reduction relation* $\twoheadrightarrow_R$ is the reflexive, transitive closure of the one-step reduction relation $\to_R$:

$$\frac{M \to_R N}{M \twoheadrightarrow_R N}$$

$$M \twoheadrightarrow_R M$$

$$\frac{M \twoheadrightarrow_R N \quad N \twoheadrightarrow_R L}{M \twoheadrightarrow_R L}$$

Definition 19. R-convertibility
R-convertibility, also called *R-equality*, is the equivalence relation generated by $\twoheadrightarrow_R$:

$$\frac{M \twoheadrightarrow_R N}{M =_R N}$$

$$\frac{M =_R N}{N =_R M}$$

$$\frac{M =_R N \quad N =_R L}{M =_R L}$$

One might think that β-reduction always leads to shorter terms, but consider the following example:

$$\begin{aligned}(\lambda x.xxy)(\lambda x.xxy) &\to_\beta (\lambda x.xxy)(\lambda x.xxy)y \\ &\to_\beta (\lambda x.xxy)(\lambda x.xxy)yy \\ &\to_\beta (\lambda x.xxy)(\lambda x.xxy)yyy \\ &\to_\beta \dots\end{aligned}$$

In this case, we say that the term $(\lambda x.xxy)(\lambda x.xxy)$ *diverges.*

Proposition 2. *$\to_R$, $\twoheadrightarrow_R$ and $=_R$ are compatible.*

Simliar to the previous Substitution Lemma, we have:

Lemma 4. *$N \twoheadrightarrow_R N' \implies M[x := N] \twoheadrightarrow_R M[x := N']$*

Definition 20. R-redex
Let $M \in \Lambda$. M is called an *R-redex* **iff** $\exists N \in \Lambda : (M, N) \in R$. N is called an R-contractum of M.
M is called an R-normal form (R-nf) if it does not contain any R-redex.
N is an R-nf of M if N is an R-nf and $M =_R N$.

3.5. **The Church-Rosser Theorem.** We now come to a central result of the λ-calculus, the *Church-Rosser Theorem*, but first we need two more definitions:

Definition 21. The Diamond Property
Let $\rhd$ be a binary relation on Λ. $\rhd$ satisfies the *diamond property*, written $\rhd \models \diamondsuit$, if:

$$\forall M, M_1, M_2 : (M \rhd M_1 \land M \rhd M_2 \implies \exists M_3 : (M_1 \rhd M_3 \land M_2 \rhd M_3))$$

This definition states that if $\rhd$ satisfies the diamond property, then, if there are two diverging $\rhd$-steps from some term, there is always a way to converge again.

Definition 22. Church-Rosser
A notion of reduction is said to be *Church-Rosser (CR)* if $\twoheadrightarrow_R \models \diamondsuit$.

This definition leads us to the following useful corrolary:

Corollary 1. *Let R be CR. Then:*
(1) *if N is an R-nf of M then $M \twoheadrightarrow_R N$*
(2) *a term can have at most one R-nf*

The proof for this corollary and the following proposition can be found in [5].

Proposition 3. *β is CR*

This means that:
(1) β-nfs are unique
(2) if a term N has a β-nf, it is possible to reduce to it.

A visual way to express the CR-property of β is to use commutative diagrams. Let $M \in \Lambda$ and $M \twoheadrightarrow_\beta M_1$ and $M \twoheadrightarrow_\beta M_2$:

$$\begin{tikzcd}
 & M \arrow[dl, "\twoheadrightarrow_\beta"'] \arrow[dr, "\twoheadrightarrow_\beta"] & \\
M_1 & & M_2
\end{tikzcd}$$

Because β is CR, there exists an $N \in \Lambda$ so that:

$$\begin{tikzcd}
 & M \arrow[dl, "\twoheadrightarrow_\beta"'] \arrow[dr, "\twoheadrightarrow_\beta"] & \\
M_1 \arrow[dr, "\twoheadrightarrow_\beta"'] & & M_2 \arrow[dl, "\twoheadrightarrow_\beta"] \\
 & N &
\end{tikzcd}$$

4. Computability

We will now put the λ-calculus to use as a tool for formulating computational problems. First we will examinine a very useful and perplexing combinator, build up an assortment of tools for computation, and finally we will close with an important result connecting the λ-calculus to other formalisms.

4.1. **Why?**
A fundamental, and at first surprising, theorem of the λ-calculus is the following:

Theorem 1. *The Fixed Point Theorem*
For all $F \in \Lambda$, there exists an $X \in \Lambda$ so that $FX = X$

Proof. Let $W \equiv \lambda x.F(xx)$ and $X \equiv WW$. Then:

$$X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) \equiv FX$$

□

This theorem states that for every F there is an X that, applied to F, gives a result that is convertible with X.

An even more surprising discovery is generally attributed to H. Curry:

Definition 23. The $\mathbf{Y}$ combinator
$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

This term, applied to another term, equals the fixed point of that term. (Keenan named it the *Why* bird, appropriately.)

For example, if we want to calculate the fixed point of $\mathbf{KI} \equiv \lambda xy.y$:

$$\begin{aligned}
\mathbf{YKI} &\equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))(\lambda xy.y) \\
&\twoheadrightarrow_\beta (\lambda x.(\lambda xy.y)(xx))(\lambda x.(\lambda xy.y)(xx)) \\
&\twoheadrightarrow_\beta (\lambda x.(\lambda y.y))(\lambda x.(\lambda y.y)) \\
&\equiv (\lambda xy.y)(\lambda xy.y) \\
&\twoheadrightarrow_\beta \lambda y.y
\end{aligned}$$

Let us test if this is really a fixed point of $\lambda xy.y$:

$$(\lambda xy.y)(\lambda y.y) \twoheadrightarrow_\beta (\lambda y.y)$$

The $\mathbf{Y}$ combinator seems to work. *Why?*

4.2. **The λ-calculus as a programming language.** To be able to formulate computational problems, we (re-)introduce some combinators and give them familiar names:

Definition 24. Boolean Values

$$\begin{aligned}\mathbf{true} &\equiv \lambda xy.x \equiv \mathbf{K}\\ \mathbf{false} &\equiv \lambda xy.y \equiv \mathbf{KI}\end{aligned}$$

So **true** is just the familiar **K** combinator, which takes two arguments and returns (reduces to) the first one. Conversely, **false** takes two arguments and returns the second one. How do these combinators work as boolean values? To see this, we introduce another combinator, which will serve as our conditional:

Definition 25. Conditional
$\mathbf{test} \equiv \lambda xyz.xyz$

test takes three arguments, the first of which should be either **true** or **false**, and applies the boolean value to the following pair of arguments, corresponding to the familiar `if x then y else z` of many programming languages.

For example, if we set x to be **false**, we get:

$$\begin{aligned}\mathbf{test\ false}\ uv &\equiv (\lambda xyz.xyz)(\lambda xy.y)yz\\ &\equiv_\alpha (\lambda xyz.xyz)(\lambda xv.v)yz\\ &\twoheadrightarrow_\beta (\lambda yz.(\lambda xv.v))yz\\ &\twoheadrightarrow_\beta (\lambda z.(\lambda xv.v)y)z\\ &\twoheadrightarrow_\beta (\lambda xv.v)yz\\ &\twoheadrightarrow_\beta (\lambda v.v)z\\ &\twoheadrightarrow_\beta z\end{aligned}$$

Which is what we would also expect the result of `if false then y else z` to be.

Definition 26. Pairs
The pairing operation, $[\_,\_]$, is defined as:

$$[M,N] \equiv \lambda z.zMN$$

Definition 27. Standard Numerals

$$\begin{aligned}\langle 0\rangle &\equiv \mathbf{I}\\ \langle n+1\rangle &\equiv [\mathbf{F},\langle n\rangle]\end{aligned}$$

For example, $\langle 3\rangle \equiv [\mathbf{F},[\mathbf{F},[\mathbf{F},\mathbf{I}]]]$.

The successor and predecessor functions are defined as:

$$\begin{aligned}\mathbf{S}^+ &\equiv \lambda x.[\mathbf{F},x]\\ \mathbf{P}^- &\equiv \lambda x.x\mathbf{F}\end{aligned}$$

We can also define a predicate that tests for $\langle 0\rangle$:

$$\mathbf{Zero} \equiv \lambda x.x\mathbf{T}$$

With these and some additional results (see [5] and [7]), we can state that *every partial recursive function can be represented by a* λ*-term* and thus λ*-definability captures the notion of effective calculability.*

We close this paper with a result the emphasizes the power of the λ-calculus:

$$\boxed{\phi \text{ is partial recursive} \Leftrightarrow \phi \text{ is } \lambda\text{-definable} \Leftrightarrow \phi \text{ is Turing Computable}}$$

References

1. H.P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, 2nd ed., North-Holland, 1984.
2. E. Cheng, *Mathematical morality*,
`http://math.unice.fr/~eugenia/morality/`.
3. A. Church, *The Calculi of Lambda-conversion*, Princeton University Press, 1941.
4. G. Goos and W. Zimmermann, *Vorlesungen über Informatik, Grundlagen und funktionales Programmieren*, 3nd ed., eXamen.press, vol. 1, Springer, 2006.
5. C. Hankin, *An Introduction to Lambda Calculi for Computer Scientists*, 2nd ed., Texts in Computing, vol. 2, King's College Publications, 2004.
6. D. Keenan, *To Dissect a Mockingbird, A Graphical Notation for the Lambda Calculus with Animated Reduction*,
`http://users.bigpond.net.au/d.keenan/Lambda/`.
7. G. Mazzola, G. Milmeister, and J. Weissmann, *Comprehensive Mathematics for Computer Scientists*, 1st ed., Universitext, vol. 2, Springer, 2005.
8. P. Wadler, *Faith, Evolution, and Programming Languages*,
`http://video.google.com/videoplay?docid=-4167170843018186532`.
9. C. Zimmer, *The Loom: Mathematical Markings*,
`http://scienceblogs.com/loom/2007/09/21/mathematical_markings.php`.

E-mail address: `therzog@cosy.sbg.ac.at`
URL: `http://alphameta.net/research`