Formal Methods
2. Lambda Calculus

Nachum Dershowitz

March 2000

The lambda calculus was designed as a general theory of (comptable) functions.

Lambda expressions are constructed from formal parameters, applications (denoted by juxtaposition) AB, where A and B are lambda expressions, and function abstractions of the form $\lambda v.A$ where v is a formal parameter and A is a lambda expression in which occurrences of v are *bound*.

The λ-calculus consists of the axioms of equality plus the following schema:

$$(\lambda v.A[v])B \quad = \quad A[B]$$

Here A and B are arbitrary λ expressions and $A[B]$ is A with each free occurrence of the parameter v replaced by B. More formally, one can define substitutions, like $\{v \mapsto B\}$ and their application to expressions, $A\{v \mapsto B\}$, replacing all free occurences of v in A with B.

As a rewrite rule, we use beta-reduction:

$$\beta : \quad (\lambda v.A[v])B \quad \rightarrow \quad A[B]$$

Theorem 1 *Beta-reduction is Church-Rosser.*

The reason is basically the same as for orthogonal systems (think of v as a constant symbol, B as a variable, and both A and $A[B]$ as schemata). Thus, $A = B$ is a theorem of the lambda calculus iff $A \downarrow B$. But β is not terminating, so not all lambda expressions have normal forms.

Theorem 2 *The lambda calculus is* consistent *in the sense that one cannot prove every closed equation.*

A *redex* is a subterm at which a rewrite rule applies (β in our case).

Theorem 3 (Leftmost Normalization) *The normal form of any expression having one can be computed by repeated application of β to the* leftmost *redex.*

Proof Suppose $A \rightarrow^{\ell} B$ at the leftmost redex, and $A \rightarrow C$ at an arbitrary redex. Then, $C \rightarrow^{\ell} D$ for some D such that $B \rightarrow^{\|} D$, where $\rightarrow^{\|}$ signifies parallel reduction. It follows that were there an infinite leftmost computation, then there could not be any normalizing computation. □

Functions of multiple arguments can be "Curried" so that only lambda abstraction with one formal parameter are needed. Hence, we will use $\lambda x_1 \ldots x_n.A$ as an abbreviation for $\lambda x_1.(\cdots(\lambda x_n.A)\cdots)$.

Lambda expressions and beta-reduction provide a (Turing-) complete model of computation. In particular, (partial) recursive functions over the natural numbers can be simulated by lambda expressions. Arithmetic and logical operations are simulated as in the following table:

$\mathbf{T}$	$\lambda uv.u$
$\mathbf{F}$	$\lambda uv.v$
$\mathbf{if}\ x\ \mathbf{then}\ y\ \mathbf{else}\ z$	$(xy)z$
$\mathbf{cons}(y,z)$	$\lambda u.(\mathbf{if}\ u\ \mathbf{then}\ y\ \mathbf{else}\ z)$
$\mathbf{car}(x)$	$x\mathbf{T}$
$\mathbf{cdr}(x)$	$x\mathbf{F}$
0	$\lambda v.v$
$x+1$	$\mathbf{cons}(\mathbf{F},x)$
$x=0$	$x\mathbf{T}$

Recursion is effected by the following mechanism: Suppose

$$f(x) \quad \stackrel{!}{=} \quad A[f]$$

where A is the **body** of the definition, containing recursive calls to f. The expression

$$(\lambda v.(A[v](vv))(\lambda v.(A[v](vv))$$

computes f.

The lambda calculus as described above does not fully capture the notion of equality of functions. For one thing, the names of parameters are immaterial: $\lambda u.u$ is the same identity function as $\lambda v.v$. Considering expressions equal if they are teh same except for parameter renaming is called α-conversion. A more serious deficiency is that $\lambda v.Av$ and A are not convertible, though for all x one has

$$Ax \downarrow (\lambda v.Av)x$$

In general, one may want to infer equality by extensionality:

$$\frac{\forall x.Ax = Bx}{A = B}$$

Definition 1 *A lambda expression of the form* $\lambda x_1 \ldots x_n.(x(\cdots(A_1 A_2)\cdots A_k))$ *is called a* head normal form.

Head normal forms are not unique.