UNIVERSITY OF CALIFORNIA, LOS ANGELES
Computer Science Department

NOTES on LAMBDA CALCULUS

David F. Martin
Fall 1992

INTRODUCTION

Lambda calculus, invented by Alonzo Church in the 1930s, is a general but syntactically simple model of computation. It was conceived as part of a system of higher-order logic and function theory. The first undecidability results were for lambda calculus; similar results for Turing machines came later. In addition to its purely mathematical applications, lambda calculus is important in the study of computer programming languages. It has served as a basic linguistic prototype from which LISP, ALGOL-like languages, and functional programming languages have been derived. It also serves as a basic metalanguage for expressing the denotational semantics of programming languages.

These notes are a brief introduction to the *type-free* lambda calculus. Two versions of the type-free lambda calculus are presented: the *call-by-name* (CBN) and the *call-by-value* (CBV) calculi. The CBN calculus was the original version of the lambda calculus; the CBV calculus is a related derivative motivated by computer science applications. These lambda calculi share the same syntax, and the CBV calculus was formulated to have the desirable properties of the CBN calculus, the most notable of these being the satisfaction of the *Church-Rosser Theorem*.

Much of the material in these notes was gleaned from Plotkin (1975). Encylopedic references are Barendregt (1984) and Hindley and Seldin (1986). An introduction to the type-free lambda calculus is Barendregdt (1977). Introductions to typed lambda calculus can be found in Hindley and Seldin (1986) and Revesz (1988). An historical account of the development of the lambda calculus is given in Rosser (1984). Davis (1989) contains a very nice account of the relation between the deductive and computation aspects of lambda calculus. Peyton-Jones (1987) is an excellent reference on functional programming languages; Chapters 8 and 9 (written by P. Hancock) provide an introduction to type checking. An excellent and detailed tutorial on how to implement functional programming languages is provided by Peyton-Jones and Lester (1992). References on lambda calculus and functional programming are given in the concluding section (References) of these notes.

THE SYNTAX OF LAMBDA NOTATION

Lambda notation is a useful technique for writing function denotations and expressing their application to actual arguments. Two syntaxes are commonly used in practice: an unambiguous one, sometimes used by mathematicians, and an ambiguous one, used both in mathematics and computer science. The syntax definitions below are written as context-free grammars; they generate the syntax domain **Term** of lambda terms, simply called "terms" for brevity. Two primitive syntax domains are assumed: **Var**, an infinite set of *variables*, and **Const**, a (not necessarily infinite) set of *constants*. In the following grammar, *variable* $\in$ **Var**, *constant* $\in$ **Const**, and terms are generated by the nonterminal *term* $\in$ **Term**.

	Syntax of Lambda Terms
U1	$term ::= constant$
U2	$term ::= variable$
U3	$term ::= (\, term \;\; term \,)$
U4	$term ::= \lambda \; variable \; . \; term$

A term constructed by rule U3 is called a *combination*, and is intended to represent function application. A combination's left term is called its *operator*, and the right term is called its *operand*. A term constructed by rule U4 is called an *abstraction*, and is intended to represent a function denotation. An abstraction's variable is called its *bound variable* (the function's formal parameter), and the term is called its *body*. A term is a *value* if and only if it is *not a combination*.

Function application is often written fx instead of $(f\ x)$ (or $f(x)$). Using juxtaposition to denote function application introduces ambiguity that can be removed explicitly by using parentheses or implicitly by assuming that function application *associates to the left*: fgx means $((f\ g)\ x)$.

The (multilayer) abstraction $\lambda x_1.\cdots\lambda x_n.M$ is often written as $\lambda x_1\cdots x_n.M$. Infix binary function application is often informally used: $x + y$, $x = 0$, etc., where (for example) $x + y$ informally represents $((+\ x)\ y)$.

Two terms M and N are *identical*, written $M \equiv N$, if they are, symbol for symbol, *exactly the same*.

Subterms and Contexts

Any term is a *subterm* of itself. In addition, a combination's operator and operand are each a subterm of that combination, and an abstraction's body is a subterm of that abstraction. M is a *proper subterm* of N if M is a subterm of N and $M \not\equiv N$.

A *context*, denoted $C[\,]$, is a term from which one subterm has been deleted. The deleted subterm is denoted by a pair $[\,]$ of brackets. An example context is $C[\,] \equiv \lambda x.(x\ [\,])$. The result of replacing the missing subterm of a context $C[\,]$ with a term M is denoted $C[M]$. In the foregoing example, $C[(x\ y)] \equiv \lambda x.(x\ (x\ y))$.

FREE AND BOUND VARIABLES

Each term X has a set $FV(X)$ of *free variables* and a set $BV(X)$ of *bound variables*, defined by induction on the structure of X as follows. Let M, N be terms, $x \in$ **Var**, and $a \in$ **Const**.

X	$FV(X)$	$BV(X)$
a	$\emptyset$	$\emptyset$
x	$\{x\}$	$\emptyset$
$(M\ N)$	$FV(M) \cup FV(N)$	$BV(M) \cup BV(N)$
$\lambda x.M$	$FV(M) - \{x\}$	$BV(M) \cup \{x\}$

A term X is *closed* iff $FV(X) = \emptyset$; otherwise it is *open*. Closed terms are sometimes called *combinators*.

PROPER SUBSTITUTION

Let $\mathcal{V} = < x_1, x_2, \ldots >$ be an infinite list of variables. Let X, M, N be terms, $x, y \in$ **Var**, and $a \in$ **Const**.

Proper substitution of M for all free occurrences of x in X, written $X[M/x]$, is defined inductively on the structure of X as follows.

X	$X[M/x]$
a	a
x	M
$y,\ y \not\equiv x$	y
$(N_1\ N_2)$	$(N_1[M/x]\ N_2[M/x])$
$\lambda x.N$	$\lambda x.N$
$\lambda y.N,\ y \not\equiv x$	$\lambda z.((N[z/y])[M/x])$

where z is the variable defined by:

(1) if $x \notin FV(N)$ or $y \notin FV(M)$, then $z \equiv y$;

(2) otherwise, z is the first variable in $\mathcal{V}$ such that $z \notin FV(N) \cup FV(M)$.

Note that $\lambda z.((N[z/y])[M/x])$ reduces to $\lambda y.(N[M/x])$ when $z \equiv y$. The restrictions in (1) and (2) prevent free variables of a term substituted into the body of an abstraction from becoming bound variables in that abstraction. This would improperly change the semantics of the new abstraction, and would render the lambda calculus *inconsistent* (it could then be deduced in λ_N (see below) that $M = N$ for *any* two terms M and N). Such an improper change of free variables into bound variables is called *capture* (of free variables).

A structure of the form $[M/x]$, where M is a term and x a variable, is called a *substitution*. Note that a substitution applied to a term is a unary *suffix* operator. Substitutions θ_i, unlike ordinary functions (which are applied as *prefix* operators), associate to the *left*, i.e., $M\theta_1\theta_2 = (M\theta_1)\theta_2$ and the members of a sequence $\theta_1\theta_2 \cdots \theta_n$ of substitutions applied to a term are applied from *left to right*. Some authors define substitution as a unary *prefix* operator, in which case substitutions associate to the *right*.

THE CALL-BY-NAME (CBN) LAMBDA CALCULUS λ_N

The call-by-name (CBN) lambda calculus λ_N is given by a formal system (that looks like a formal logic, with axioms and inference rules) that defines a *conversion relation* $=_N$ (sometimes written simply as $=$ when no confusion results) between terms. λ_N is a formal theory of *CBN lambda equality*. To avoid confusion with ordinary equality, $=$ is sometimes written *cnv* (denoting conversion), as in M *cnv* N.

Let $x, y \in$ **Var**; let $a, b \in$ **Const**; let M, N, Z be terms.

	CBN Lambda Calculus λ_N
I(α)	$\lambda x.M = \lambda y.(M[y/x])$, provided that $y \notin FV(M)$
I(β)	$(\lambda x.M\ N) = M[N/x]$
I(δ)	$(a\ b) = ConstApply(a, b)$, if defined
II(1)	$M = M$
II(2)	$\dfrac{M = Z,\ Z = N}{M = N}$
II(3)	$\dfrac{M = N}{N = M}$
III(1)	(a) $\dfrac{M = N}{(M\ Z) = (N\ Z)}$, (b) $\dfrac{M = N}{(Z\ M) = (Z\ N)}$
III(2)	$\dfrac{M = N}{\lambda x.M = \lambda x.N}$

The rules in group I are the basic *conversion rules* of λ_N. Rule I(α), called *α-conversion*, specifies the *renaming, or change, of bound variables* in a term. The restriction $y \notin FV(M)$ prevents capture of free occurrences of y in M by λy. Rule I(β) specifies *contraction* (also called *β-reduction*) and rule I(δ) specifies the corresponding operation of *constant contraction* (also called *δ-reduction*). The *partial* function

$$ConstApply : \mathbf{Const} \times \mathbf{Const} \xrightarrow{P} \mathbf{ClosedValues},$$

where **ClosedValues** denotes the set of closed terms that are also values, specifies how to reduce combinations in which both components are constants. The term $(\lambda x.M\ N)$ in rule I(β) is called a *CBN β-redex* (**red**ucible **ex**pression). In rule I(δ), the term $(a\ b)$, *when* $ConstApply(a,b)$ *is defined*, is called a *CBN* δ*-redex*.

The rules in group II assert that $=$ is reflexive, transitive, and symmetric, i.e., that $=$ is an *equivalence relation*. The rules in group III make $=$ *substitutive*.

λ_N can be viewed as a formal deductive system in which the well-formed formulas (WFFs) have the form $M = N$, where M and N are terms. $\lambda_N \vdash M = N$ means that $M = N$ is provable by the rules of λ_N, i.e., $M = N$ is a *theorem* of λ_N. $\lambda_N \vdash M \rhd N$ means that $M = N$ is provable by the rules of λ_N *without using II(3)*. It is important to keep in mind that the relation $=$, despite its notational similarity to equality, is reflexive, transitive, and symmetric *only because of rules II(1)–(3) in* λ_N. If these rules were absent, then $=$ would not necessarily have these properties. Thus $\rhd$ is reflexive and transitive, but *not* symmetric.

Strictly speaking, $=$ and $\rhd$ should be written $=_N$ and $\rhd_N$ to denote that they are associated with λ_N, and this will be done when necessary to avoid confusion with other, similar, relations defined by other, similar, formal systems.

$\rhd$ is called a *reduction relation*; it expresses the reduction, or *simplification*, of a term into a usually simpler or shorter term. Although rules I(α)–I(δ) all contribute to $\rhd$, rules I(β) and I(δ) are the ones that actually have the potential for term simplification, since I(α) only renames a term's bound variables, which preserves the structure of that term. When $M \rhd N$, we say that M is *reduced to* N.

THE CALL-BY-VALUE (CBV) LAMBDA CALCULUS λ_V

The call-by-value (CBV) lambda calculus λ_V is given by a formal system that defines a *conversion relation* $=_V$ and a *reduction relation* $\rhd_V$. λ_V is a theory of *CBV lambda equality*. λ_V is the same as λ_N except for a restriction on rule I(β):

	CBV Lambda Calculus λ_V
I(β)	$(\lambda x.M\ N) = M[N/x]$, provided that N is a *value*
	Rules I(α), I(δ), II(1)–II(3), and III(1)–III(2) are the same as those in λ_N.

Recall that a value is a term that is *not a combination*. $\lambda_V \vdash M = N$ means that $M = N$ is provable by the rules of λ_V. $\lambda_V \vdash M \rhd N$ means that $M = N$ is provable by the rules of λ_V *without using II(3)*.

SUBTERM REPLACEMENT

The relations $M = N$ and $M \rhd N$ (converting or reducing a term to another term) in λ_N and λ_V are established by *formal deductions* in those systems. Establishing these formal deductions can be a lengthy and tedious process. It would be easier to convert (reduce) a term to another term by simply replacing one of its subterms that is also a redex by another subterm to which that redex converts (reduces). Indeed, all of the conversion (reduction) is done by the axioms in group I. The rules in groups II and III simply distribute these conversions (reductions) over the components of combinations and the bodies of abstractions.

This conversion (reduction) strategy is justified by a *derived rule* of λ_N (and λ_V) called *subterm replacement* (also called the *substitution rule*), stated using the notion of contexts:

	Subterm Replacement
IV	$\dfrac{M = N}{C[M] = C[N]}$, for all contexts $C[\,]$

NORMAL FORMS

A term is in *CBN (CBV) normal form (NF)* if none of its subterms is a CBN (CBV) β- or δ- redex. A term is said to *have a NF* if it can be reduced to a NF.

It is important to note that not every term has a NF. For example, $(\lambda x.(x\ x)\ \lambda x.(x\ x))$ does not have a NF in λ_N or λ_V because it is both a CBN and CBV β-redex that cannot be converted or reduced to anything but itself in λ_N and λ_V.

Unfortunately, even if a term has a NF, it may not be possible to determine it: *there exists no algorithm to determine whether or not an arbitrary term has a NF.*

THE CHURCH-ROSSER THEOREM

Usually one term can be reduced, or simplified, to another via $\rhd$ in a variety of ways. This is because a term may contain several redexes, any one of which can be chosen to be contracted next by an application of I(β) or I(δ). It is natural, and important, to ask whether the choice of redex on which to carry out an individual reduction makes any "difference" in the outcome of a sequence of individual reductions applied to reduce one term to another. The Church-Rosser Theorem (CRT) and a Corollary say that different sequences of choices yield results that are "essentially the same" in a precise sense defined below. The CRT applies to both λ_N and λ_V. In the statement of the theorem, λ represents either λ_N or λ_V throughout.

The Church-Rosser Theorem

Let L, M_1, M_2, and N be terms. If $\lambda \vdash L \rhd M_1$ and $\lambda \vdash L \rhd M_2$, then there exists N such that $\lambda \vdash M_1 \rhd N$ and $\lambda \vdash M_2 \rhd N$. $\Box$

Terms M and N are *alphabetically equivalent*, written $M \equiv_\alpha N$, if $\lambda \vdash M = N$ *without using rules I(β)–(δ)*, where λ represents either λ_N or λ_V. In other words, two terms are alphabetically equivalent if they differ only in the names of their bound variables. Alphabetic equivalence is clearly an equivalence relation.

Corollary (to the Church-Rosser Theorem)

If $\lambda \vdash L \rhd M_1$ and $\lambda \vdash L \rhd M_2$ and M_1 and M_2 are both in NF, then $M_1 \equiv_\alpha M_2$. $\Box$

The above corollary states that if a term can be reduced to different NFs, then these NFs are alphabetically equivalent, i.e., they are the same except for the names of their bound variables. The CRT and its Corollary state that if a term has a NF, then that NF is *unique* (up to the names of its bound variables).

In λ_V, the restriction that N be a *value* in I(β) is essential to ensure that the CRT holds for λ_V. If, as might be natural to suggest, this restriction were changed so that N is required to be a *CBV NF* instead of a value, then the CRT fails for such a modified version of λ_V. An example of such a failure is provided by

$$L = (\lambda x.(\lambda y.z\ (x\ \lambda x.(x\ x)))\ \lambda x.(x\ x))$$
$$M_1 = z$$
$$M_2 = (\lambda y.z\ (\lambda x.(x\ x)\ \lambda x.(x\ x)))$$

Thus $\lambda_V \vdash L \rhd M_1$ and $\lambda_V \vdash L \rhd M_2$, but since M_1 is in NF and M_2 cannot be further reduced to anything but itself, there exists no term N such that $\lambda_V \vdash M_1 \rhd N$ and $\lambda_V \vdash M_2 \rhd N$. In the *original* version of λ_V, $\lambda_V \vdash L \rhd M_2$, but $\lambda_V \not\vdash L \rhd M_1$ because $(x\ \lambda x.(x\ x))$ is not a value.

STANDARD REDUCTION STRATEGIES

The CRT and its corollary state that if a term can be reduced to NF in two different ways, then the (possibly different) NFs obtained are alphabetically equivalent, and so the particular order of reduction chosen doesn't make any *essential* difference in the resulting NF. There is, however, a "standard" way of choosing a next redex to contract that ensures the reduction of a term to NF, provided that the term actually has a NF.

In what follows, the *unambiguous* syntax of λ-terms is assumed. A redex that is a subterm of a term is the *leftmost* redex of that term if the leftmost symbol of that redex is located to the left of the leftmost symbol of any other redex that is a subterm of the term. A *normal order reduction sequence* is a sequence of terms in which the last element is a NF of the first element, and in which each element is obtained from the immediately previous element by contracting the previous element's *leftmost* β- or δ-redex. A *normal order reduction strategy* (NORS) is one whereby a term is reduced to NF via a normal order reduction sequence.

The Standardization Theorem

If a term has a NF, then that NF can always be obtained (to within alphabetic equivalence) by reducing the term via a NORS. □

The Standardization Theorem applies to both λ_N and λ_V. In λ_N, a normal order reduction strategy is often called a *leftmost outermost* reduction strategy. The word "outermost" is redundant if the unambiguous syntax of terms is assumed: the leftmost redex is also an outermost one. The NORS is often called the "call-by-name" reduction strategy. Still within λ_N, there is another reduction strategy, the *applicative order reduction strategy* (AORS), in which the *leftmost innermost* β- or δ-redex is contracted. The AORS is often called the "call-by-value" reduction strategy (not to be confused with λ_V) because it forces the operand of a β-redex to be reduced to NF before that β-redex can be contracted.

If a term has a NF then the NORS will always obtain it (to within alphabetic equivalence), whereas the AORS may not. For example, let $\Delta \equiv \lambda x.(x\ x)$. Then

$$\text{(NORS)} : (\lambda y.x\ (\Delta\ \Delta)) \rhd x \text{ [converges to NF]}$$
$$\text{(AORS)} : (\lambda y.x\ (\Delta\ \Delta)) \rhd (\lambda y.x\ (\Delta\ \Delta)) \rhd \cdots \text{ [diverges]}$$

η-REDUCTION

Because it provides a reasonable and desirable way to help simplify terms, an additional rule, called η-*reduction*, is often included in λ_N:

	η-Reduction
I(η)	$\lambda x.(M\ x) = M$, provided that $x \notin FV(M)$.

Rule I(η) is a *cancellation rule* that permits simplifications such as $\lambda x.f(x) = f$. The restriction $x \notin FV(M)$ is necessary; its omission would render λ_N *inconsistent*. If x were free in M, then reducing $\lambda x.(M\ x)$ to M would be intuitively (and semantically) invalid because x is not free in $\lambda x.(M\ x)$ whereas x is left "floating" free in M after the reduction.

Rule I(η) combines with rule III(2) to yield the *Extensionality Principle*:

	Extensionality Principle
V	$\dfrac{(M\ x) = (N\ x)}{M = N}$, provided that $x \notin FV(M) \cup FV(N)$.

The Extensionality Principle expresses the lambda calculus form of the notion of *extensional equality* of functions: $f = g$ iff $(\forall x)(f(x) = g(x))$.

Unfortunately, if I(η) is included in λ_V, the CRT fails for λ_V, as in

$$L = (\lambda x.y\ \lambda x.((\lambda x.(x\ x)\ \lambda x.(x\ x))\ x))$$
$$M_1 = y$$
$$M_2 = (\lambda x.y\ (\lambda x.(x\ x)\ \lambda x.(x\ x)))$$

$L \rhd M_1$ via I(β) whereas $L \rhd M_2$ via I(η); this latter reduction could not be made without I(η). The problem with including I(η) in λ_V is that this rule can convert a term that is a value into one that is no longer a value. This occurred in $L \rhd M_2$.

THE CONSISTENCY OF λ_N AND λ_V

The CRT plays a vital role in establishing the *consistency* of λ_N and λ_V as deductive systems. λ_N (λ_V) is *consistent* if there exists a WFF $M = N$ such that $\lambda_N \not\vdash M = N$ ($\lambda_V \not\vdash M = N$), i.e., $M = N$ is *not* a theorem of λ_N (λ_V). Equivalently, λ_N (λ_V) is *inconsistent* if $\lambda_N \vdash M = N$ ($\lambda_V \vdash M = N$) for *all* WFFs $M = N$, i.e., $M = N$ is a theorem of λ_N (λ_V) for *any* terms M and N.

Theorem: λ_N and λ_V are consistent.

Proof. Suppose that λ_N were not consistent. Then for *all* terms M and N, $\lambda_N \vdash M = N$. In particular, $\lambda_N \vdash \lambda x.\lambda y.x = \lambda x.\lambda y.y$. But both $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$ are in NF and $\lambda x.\lambda y.x \not\equiv_\alpha \lambda x.\lambda y.y$, which contradicts the Corollary to the CRT. The proof is similar for λ_V. □

COMPUTATION VERSUS DEDUCTION IN λ_N AND λ_V

The reduction of a term to NF can be viewed as a "computation" that terminates in the sense that the term cannot be further simplified by applications of rules I(β)–(δ) (and I(η) if present in λ_N). In this sense, it seems reasonable to regard the attempted reduction of a term that does not have a NF as leading to a "nonterminating" computation, and therefore in this computational sense, such a term can be regarded as representing "undefined". Continuing along this line, it seems reasonable to regard *every* term that does not have a NF as representing "undefined" and thus all terms not having a NF are *identified* (*declared* to be interconvertible, i.e., related by =) in λ_N (and λ_V). This can be accomplished by adding *axioms* $M = N$ to λ_N (λ_V) when M and N do not have a CBN (CBV) NF. Let λ'_N (λ'_V) denote λ_N (λ_V) augmented with these axioms. Unfortunately, λ'_N and λ'_V are *inconsistent*.

Theorem: λ'_N is inconsistent.

Proof [Davis (1989)]. Let $\mathbf{tt} \equiv \lambda x.\lambda y.x$ and $\mathbf{ff} \equiv \lambda x.\lambda y.y$. $\mathbf{tt}$ and $\mathbf{ff}$ are intended to be representations of *true* and *false*, respectively, in λ'_N. Let U be any term that does not have a NF and let $Q \equiv \lambda x.\,((x\ \mathbf{tt})\ U)$ and $R \equiv \lambda x.\,((x\ \mathbf{ff})\ U)$. First of all, note that neither Q nor R has a NF because U doesn't. Thus $\lambda'_N \vdash Q = R$. But

$$(Q\ \mathbf{tt}) \rhd ((\mathbf{tt}\ \mathbf{tt})\ U) \rhd (\lambda y.\mathbf{tt}\ U) \rhd \mathbf{tt}$$
$$(R\ \mathbf{tt}) \rhd ((\mathbf{tt}\ \mathbf{ff})\ U) \rhd (\lambda y.\mathbf{ff}\ U) \rhd \mathbf{ff}$$

and thus $\lambda'_N \vdash (Q\ \mathbf{tt}) = \mathbf{tt}$ and $\lambda'_N \vdash (R\ \mathbf{tt}) = \mathbf{ff}$. Since $\lambda'_N \vdash Q = R$, $\lambda'_N \vdash (Q\ \mathbf{tt}) = (R\ \mathbf{tt})$ by subterm replacement, and thus $\lambda'_N \vdash \mathbf{tt} = \mathbf{ff}$ by transitivity of =. Now let M and N be *any* two terms. It is easily shown that $\lambda'_N \vdash ((\mathbf{tt}\ M)\ N) = M$ and $\lambda'_N \vdash ((\mathbf{ff}\ M)\ N) = N$ and thus since $\lambda'_N \vdash \mathbf{tt} = \mathbf{ff}$, $\lambda'_N \vdash M = N$ by subterm replacement and the transitivity of =. Thus λ'_N is inconsistent. □

The inconsistency of λ'_V is proved by the same argument.

There exist, however, "weaker" kinds of NF that do not introduce inconsistency in the above sense, called *head NF* (HNF) and *weak head NF* (WHNF) [Barendregt (1984), Field and Harrison (1988)]. In particular, a WHNF is the end result of reducing (computing upon) a term using machine models such as the SECD Machine [Field and Harrison (1988)] and practical implementations of functional programming languages [Peyton-Jones (1987)].

Using our syntax, a term is in WHNF if and only if it is a *value* (a term that is not a combination). Standard reductions of terms to WHNF (when possible) can be defined in terms of CBN and CBV *left reduction relations* $\rightarrow_N, \rightarrow_V \subseteq \mathbf{Term} \times \mathbf{Term}$, that are the least relations between terms satisfying the following conditions.

	CBN and CBV Left Reduction Relations
1N	$(\lambda x.M\ N) \rightarrow_N M[N/x]$
2N	$(a\ b) \rightarrow_N ConstApply(a,b)$ when defined
3N	$(M\ N) \rightarrow_N (M'\ N)$ if $M \rightarrow_N M'$
4N	$(M\ N) \rightarrow_N (M\ N')$ if $(M = a$ or $M = x)$ and $N \rightarrow_N N'$
1V	$(\lambda x.M\ N) \rightarrow_V M[N/x]$ when N is a value
2V	$(a\ b) \rightarrow_V ConstApply(a,b)$ when defined
3V	$(M\ N) \rightarrow_V (M'\ N)$ if $M \rightarrow_V M'$
4V	$(M\ N) \rightarrow_V (M\ N')$ if M is a value and $N \rightarrow_V N'$

Informally, if $M \rightarrow_N N$ $(M \rightarrow_V N)$, then N is obtained from M by contracting the leftmost CBN (CBV) redex of M that is not contained in the body of an abstraction.

Theorem [Plotkin (1975)]**:** For any term M and value N, $\lambda_N \vdash M \rhd N$ iff $M \xrightarrow{*}_N N$. The same result holds when λ_N and $\rightarrow_N$ are replaced by λ_V and $\rightarrow_V$. $\Box$

Rather that using λ_N and λ_V, *closed* terms can be reduced to WHNF (i.e., values) by corresponding (partial) *evaluation functions* $eval_N, eval_V$: **ClosedTerms** $\xrightarrow{P}$ **ClosedValues**, defined as follows.

CBN and CBV Evaluation Functions
$eval_N(a) = a$
$eval_N(\lambda x.M) = \lambda x.M$
$eval_N((M\ N)) = eval_N(M'[N/x])$ if $eval_N(M) = \lambda x.M'$
$eval_N((M\ N)) = ConstApply(a,b)$ if $eval_N(M) = a$ and $eval_N(N) = b$ and $ConstApply(a,b)$ is defined
$eval_V(a) = a$
$eval_V(\lambda x.M) = \lambda x.M$
$eval_V((M\ N)) = eval_V(M'[N'/x])$ if $eval_V(M) = \lambda x.M'$ and $eval_V(N) = N'$
$eval_V((M\ N)) = ConstApply(a,b)$ if $eval_V(M) = a$ and $eval_V(N) = b$ and $ConstApply(a,b)$ is defined

$eval_N(M)$ $(eval_V(M))$ yields M's CBN (CBV) WHNF (a *value*) if M has such a WHNF; otherwise, $eval_N(M)$ $(eval_V(M))$ is *undefined*. The following theorem relates the above evaluation functions (computation) to the left reduction relations and hence to the λ-calculi (deduction).

Theorem [Plotkin (1975)]**:** For any closed term M and value N, $eval_N(M) = N$ iff there exists a term N' such that $M \xrightarrow{*}_N N'$ and $N' \equiv_\alpha N$. The same result holds when $eval_N$ and $\rightarrow_N$ are replaced by $eval_V$ and $\rightarrow_V$. $\Box$

CONDITIONAL AND RECURSION IN THE LAMBDA CALCULUS

In this section, it is assumed that the *evaluation functions* $eval_N$ (for λ_N) $eval_V$ (for λ_V) are used to *compute* a (closed) term to a *value*, if possible. Alternatively, the *left reduction relations* $\rightarrow_N$ and $\rightarrow_V$ could be used to *reduce* a term to a value.

Conditional

The conditional (if-then-else) operator can be encoded in the lambda calculus by introducing constants **COND**, **true**, and **false**. Let $ConstApply_N$ and $ConstApply_V$ denote the constant application function for λ_N and λ_V, respectively. Then

$$ConstApply_N(\mathbf{COND}, \mathbf{true}) = \lambda x.\lambda y.x$$
$$ConstApply_N(\mathbf{COND}, \mathbf{false}) = \lambda x.\lambda y.y$$

$$ConstApply_V(\mathbf{COND}, \mathbf{true}) = \lambda x.\lambda y.\,(x\ a_0)$$
$$ConstApply_V(\mathbf{COND}, \mathbf{false}) = \lambda x.\lambda y.\,(y\ a_0)$$

where a_0 is an arbitrary constant. Let L, M, and N be terms. Then **if** L **then** M **else** N is encoded in λ_N and λ_V as follows:

$$(\lambda_N): \quad (((\mathbf{COND}\ L)\ M)\ N)$$
$$(\lambda_V): \quad (((\mathbf{COND}\ L)\ \lambda z.M)\ \lambda z.N) \ \text{ where } z \notin FV(M) \cup FV(N)$$

It is assumed that L can be reduced to one of the constants **true** or **false**. The "encapsulation" of M and N in $\lambda z.M$ and $\lambda z.N$ constructs a CBV conditional operator that prevents M and N from being evaluated until one of them has been selected.

Recursion

Let **rec** $x = M$ denote the *recursive* definition of x. Normally $x \in FV(M)$, but this is not necessary; if $x \notin FV(M)$, then no recursion is being defined. Define CBN and CBV *fixed point combinators* Y and Z:

$$(\lambda_N): Y \equiv \lambda f.(\lambda g.(f\ (g\ g))\ \lambda g.(f\ (g\ g)))$$
$$(\lambda_V): Z \equiv \lambda f.(\lambda g.(f\ \lambda h.((g\ g)\ h))\ \lambda g.(f\ \lambda h.((g\ g)\ h)))$$

rec $x = M$ is encoded in λ_N and λ_V as follows:

$$(\lambda_N): (Y\ \lambda x.M)$$
$$(\lambda_V): (Z\ \lambda x.M)$$

REFERENCES

Lambda Calculus

Barendregt, H. P., "The Type Free Lambda Calculus", in Barwise, J. (Ed.), *Handbook of Mathematical Logic*, North-Holland Publishing Co., Amsterdam, 1977, pp. 1091-1132.

Barendregt, H. P., *The Lambda Calculus: Its Syntax and Semantics*, 2nd Edition, North-Holland Publishing Co., Amsterdam, 1984.

Davis, R. E., *Truth, Deduction, and Computation*, Computer Science Press, New York, 1989.

Hindley, J. R. and Seldin, J. P., *Introduction to Combinators and λ-Calculus*, Cambridge University Press, Cambridge, U. K., 1986.

Plotkin, G. D., "Call-by-Name, Call-by-Value, and the λ-Calculus", *Theoretical Computer Science* 1: 125-159, 1975.

Revesz, G., *Lambda-Calculus, Combinators, and Functional Programming*, Cambridge University Press, Cambridge, U. K., 1988.

Rosser, J. B., "Highlights of the History of the Lambda-Calculus", *Annals of the History of Computing* 6: 337-349, Oct. 1984.

Functional Programming Languages

Field, A. J. and Harrison, P. G., *Functional Programming*, Addison-Wesley, Reading, Mass., 1988.

Henson, M. C., *Elements of Functional Languages*, Blackwell Scientific Publications, Oxford, U. K., 1987.

Peyton-Jones, S. L., *The Implementation of Functional Programming Languages*, Prentice Hall International, Englewood Cliffs, N. J., 1987.

Peyton-Jones, S. L. and Lester, D. R., *Implementing Functional Languages: A Tutorial*, Prentice Hall International, Englewood Cliffs, N. J., 1992.