A NEW PROGRAM FOR COMBINATORY REDUCTION AND ABSTRACTION

SUSHANT DESHPANDE

Master of Science, University of Lethbridge, 2009

A Thesis

Submitted to the School of Graduate Studies

of the University of Lethbridge

in Partial Fulfillment of the

Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science

University of Lethbridge

LETHBRIDGE, ALBERTA, CANADA

Abstract

Even though lambda calculus (λ-calculus) and combinatory logic (CL) appear to be equivalent, they are not. As yet we do not have a reduction in CL which corresponds to β-reduction in λ-calculus. There are three proposals but they all have few problems one of which is the lack of a complete characterization of CL-terms corresponding to λ-terms in β-normal form. Finding such a characterization for any of the three proposals appears to require a lot of examples which are tedious and time consuming to develop by hand. For this reason, a computer program to do reductions and abstractions of CL-terms would be useful. This thesis is about an attempt to write such a program. The program that we have does not yet work for the three proposals but it works for βη-strong reduction. Coding this program turned out to be much harder than anticipated. Dr. Robin Cockett developed a semantic translation which helped in coding the program but his semantic translation needs to be extended to all three proposals to obtain the program originally desired and that needs a lot of research.

Table of Contents

Introduction

This thesis is about the relationship between lambda calculus (λ-calculus) and combinatory logic (CL). Both the systems have the same purpose: to describe the fundamental properties of operators and combinations of operators [12, Page vi].

The λ-calculus was invented in the 1930s by an American logician named Alonzo Church, as a part of system of logic which included higher order functions. Higher-order functions are functions which can be applied to functions. The language of λ-calculus is important as a higher-order language both for logic and for programming [1, Page 10].

The basic idea of CL was presented by two logicians: Moses Schönfinkel who invented it in 1920 and Haskell Curry, who rediscovered it a few years later.

A lambda expression represents any function and defines the transformation that the function performs to its arguments. A lambda expression can be used as both a term and an argument. Here, functions are treated as first-class entities, i.e., they are passed as arguments and returned as results. λ-calculus can be thought of as an idealized, minimalistic programming language. This makes the model of functional programming important. λ-calculus is discussed in Chapter 1.

CL is a notation introduced to eliminate the need for variables in mathematical logic. It was developed to be a theory for the foundation of mathematics. Its goal was to establish fundamental mathematical concepts on simpler principals. In computer science,

combinatory logic is used as a simplified model of computation [3, Page 23]. It is also used in computability theory and proof theory. CL is discussed in Chapter 2.

As we discuss more about λ-calculus and CL, it will become clear that even though λ-calculus and CL are closely related, they are subtly different. In practice, the natural process of conversion and reduction in λ-calculus is different from that in CL.

We have combinatory β-equality and combinatory βη-equality which are equivalent to λβ-conversion and λβη-conversion respectively [Chapter 2, Page 27]. This equivalence of λ-calculus to CL is with respect to conversion but not reduction. Two main reductions in λ-calculus are the λβ-reduction and the λβη-reduction. Curry's strong reduction in CL is equivalent to λβη-reduction in λ-calculus [12, Page 213], but as of now, we do not have a complete equivalent in CL that corresponds to λβ-reduction in λ-calculus. There are a few proposals but none of them has a complete characterization of terms in normal form. This is discussed in detail in Chapter 3.

Researchers are working on solving the problem of λβ-reduction, but in order to test their theories, they need to generate a lot of examples and to reduce them. This is where a program would come in handy. The availability of a program would help them in testing their theories. I have discussed how the program was created and what difficulties we faced while writing it in chapter 4 while the actual code of the program is in appendix B. There is also a short tutorial on SML/NJ in appendix A. The current program performs strong reduction, i.e. Curry's strong reduction. As yet, it does not work for the proposals for a combinatory β-reduction.

Chapter 1

λ-calculus

1.1 Introduction:

What is usually called Lambda(λ)-calculus is a collection of several formal systems, based on a notation invented by Alonzo Church in the 1930s [12, Chapter 1]. They are designed to describe the most basic way that operators or functions can be combined to form other operators. In practice, each λ-system has a slightly different grammatical structure, depending on its intended use. Some have extra constant-symbols, and most have built-in syntactic restrictions, for example type-restrictions.

Now let us consider the everyday mathematical expression $'x - y'$. This can be considered as defining either a function f of x or a function g of y:

$$f : x \mapsto x - y, \qquad g : y \mapsto x - y.$$

There is a need for a notation that gives f and g different names in some systematic way.

Church's notation is a systematic way of constructing, for each expression involving $'x'$, a notation for the corresponding function of $'x'$ (and similarly for $'y'$, etc.). Church introduced $'\lambda'$ as an auxiliary symbol and wrote:

$$f = \lambda x.x - y \qquad g = \lambda y.x - y.$$

For example, consider the equations

$$f(0) = 0 - y, \qquad f(1) = 1 - y.$$

In the λ-notation these become

$$(\lambda x.\, x - y)(0) = 0 - y, \qquad (\lambda x.\, x - y)(1) = 1 - y.$$

The λ-notation is principally intended for denoting higher-order functions, not just functions of numbers. This notation is systematic, allowing for its incorporation into a programming language.

The λ-notation can be extended to functions of more than one variable. For example, the expression $'x - y'$ determines two functions h and k of two variables defined by

$$h(x, y) = x - y, \qquad k(y, x) = x - y.$$

These can be denoted by

$$h = \lambda xy.\, x - y, \qquad k = \lambda yx.\, x - y.$$

However, we can avoid the need of special notation for functions of several variables by using functions whose values are not numbers but other functions. For example, instead of the two-variable function h above, consider the one-place function h^* defined by

$$h^* = \lambda x.\, (\lambda y.\, x - y)$$

For each number *a*, we have

$$h^*(a) = \lambda y.\, a - y$$

Hence for each pair of numbers *a*, *b*

$$\big(h^*(a)\big)(b) = (\lambda y.\, a - y)(b)$$

$$= a - b$$

$$= h(a, b)$$

Thus h^* can be viewed as 'representing' h. This is called '*Currying*' [after H.B. Curry].

Here, following points are of significance;

(1) in λ-calculus (and in combinatory logic), it is usual to write '(*fx*)' instead of '*f*(*x*)' for the value of the function *f* at the value of *x*;

(2) for the rest of the thesis, $\lambda xy.\, xy$ will be an abbreviation for $\lambda x.\, (\lambda y.\, xy)$.

1.2 Definition (λ-terms)

Assume that there is a given infinite sequence of expressions $v_0, v_{00}, v_{000}, \ldots$ called *variables* [12, Definition 1.1], and a finite, infinite or empty sequence of expressions called *atomic constants*, different from the variables. When the sequence of atomic constants is empty, the system will be called *pure*, otherwise *applied*. The set of expressions called λ-terms is defined inductively as follows:

(a) All variables and atomic constants are λ-terms (called *atoms*);

(b) If M and N are any λ-terms, then (M N) is a λ-term (called an *application*);

(c) If M is any λ-term and x is any variable, then $(\lambda x.M)$ is a λ-term (called an *abstraction*).

Examples of λ-terms:

(a) $(\lambda v_0.(v_0 v_{00}))$ is a λ-term.

If x, y, z are distinct variables, the following are λ-terms:

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

(c) $((\lambda y.y)(\lambda x.(xy)))$

(d) $(x(\lambda x.(\lambda x.x)))$

(e) $(\lambda x.(yz))$

In example (d), there are two occurrences of λx in one term. Example (e) shows a term of form $(\lambda x.M)$ such that x does not occur in M. This is called *vacuous abstraction*, and such terms denote constant functions, i.e., functions whose output is same for all inputs.

In λ-calculus, the parentheses are left associative, that is to say that the leftmost term has the first set of parentheses.

Example:

Consider the term $xv(\lambda yz.yu)w$

This is really $(((xv)(\lambda y.(\lambda z.(yu)))w)$

Term-structure and substitution

1.3 Definition

The *length* of a term M (called $lgh(M)$) is the total number of occurrences of atoms in M.

(a) $lgh(a) = 1$ for atoms a;

(b) $lgh(M\,N) = lgh(M) + lgh(N)$;

(c) $lgh(\lambda x.M) = 1 + lgh(M)$.

'≡' implies that the term on the left hand side is identical to the term on the right hand side.

The phrase '$induction\ on\ M$' will mean 'induction on $lgh(M)$' [12, Definition 1.6].

For example, if $M \equiv x(\lambda y.yux)$ then $lgh(M) = 5$.

1.4 Definition

For λ-terms P and Q [12, Definition 1.7], the relation *P occurs in Q* (or *P is a subterm of Q*, or *Q contains P*) is defined by induction on Q, thus:

(a) P occurs in P;

(b) If P occurs in M or in N, then P occurs in (MN);

(c) If P occurs in M or $P \equiv x$, then P occurs in $(\lambda x.M)$.

1.5 Definition (Scope)

For a particular occurrence of $\lambda x.M$ in a term P [12, Definition 1.9], the occurrence of M is called the scope of the occurrence of λx on the left.

For example, assume

$$P \equiv \big(\lambda y.\, yx(\lambda x.\, y(\lambda y.\, z)x)\big)vw.$$

The scope of the left-most λy is $yx(\lambda x.\, y(\lambda y.\, z)x)$, the scope of λx is $y(\lambda x.\, z)x$, and that of the right-most λy is z.

1.6 Definition (Free and bound variables)

An occurrence of a variable x in a term P is called

- *bound* if it is in the scope of a λx in P,
- *bound and binding*, if and only if it is the x in λx,
- *free* otherwise.

If x has at least one binding occurrence in P [12 Definition 1.11], it is called a *bound variable of P*. If x has at least one free occurrence in P it is called a *free variable of P*; the set of all free variables of P is called

$$\mathrm{FV}(P).$$

A closed term is a term without any free variables.

Example: In $\int_a^b f(x,y)dx$ the variable *x* is bound and *y* is free. Hence, substituting 7 for *x*: $\int_a^b f(7,y)d7$; would be incorrect, but substitution for *y* wouldn't be: $\int_a^b f(x,7)dx$ [22].

1.7 Definition (Substitution)

For any $M, N, x,$ define $[N/x]M$ to be the result of substituting N for every occurrence of x in M [12, Definition 1.12], and changing bound variables to avoid clashes.

(a) $[N/x]x \equiv N$;

(b) $[N/x]a \equiv a$ for all atoms $a \not\equiv x$

(c) $[N/x](PQ) \equiv (([N/x]P)([N/x]Q))$;

(d) $[N/x](\lambda x.P) \equiv \lambda x.P$;

(e) $[N/x](\lambda y.P) \equiv \lambda y.P$ if $x \notin \mathrm{FV}(P)$ and $y \not\equiv x$;

(f) $[N/x](\lambda y.P) \equiv \lambda y.[N/x]P$ if $x \in \mathrm{FV}(P)$ and $y \notin \mathrm{FV}(N)$ and $y \not\equiv x$;

(g) $[N/x](\lambda y.P) \equiv \lambda z.[N/x][z/y]P$ if $x \in \mathrm{FV}(P)$ and $y \in \mathrm{FV}(N)$.

In (g), *z* is the first variable that does not occur anywhere in the term.

1.8 Lemma

For all terms M, N and variable x;

(a) $[x/x]M \equiv M$;

(b) $x \notin \mathrm{FV}(M) \Longrightarrow [N/x]M \equiv M$;

(c) $x \in \mathrm{FV}(M) \Longrightarrow \mathrm{FV}([N/x]M) = \mathrm{FV}(N) \cup (\mathrm{FV}(M) - \{x\})$;

(d) $lgh([y/x]M) = lgh(M)$.

The proof can be found in [12, Lemma 1.15, page 8].

1.9 Lemma

Let x, y, v be distinct (the usual notation convention), and let no variable bound in M be free in vPQ. Then

(a) $[P/v][v/x]M \equiv [P/x]M$ $\quad if\ v \notin \mathrm{FV}(M)$;

(b) $[x/v][v/x]M \equiv M$ $\quad if\ v \notin \mathrm{FV}(M)$;

(c) $[P/x][Q/y]M \equiv [[P/x]Q/y][P/x]M$ $\quad if\ y \notin \mathrm{FV}(P)$;

(d) $[P/x][Q/y]M \equiv [Q/y][P/x]M$ $\quad if\ y \notin \mathrm{FV}(P), x \notin \mathrm{FV}(Q)$;

(e) $[P/x][Q/x]M \equiv [[P/x]Q/x]M$.

The proof can be found in [12, Lemma 1.16, Page 9].

1.10 Definition (Change of bound variables, congruence)

Let a term P contain an occurrence of $\lambda x.M$ [12, Definition 1.17], and let $y \notin \mathrm{FV}(M)$. The action of replacing this $\lambda x.M$ by

$$\lambda y.[y/x]M$$

is called a *change of bound variable* or an *α-conversion* in *P*. If and only if *P* can be changed to *Q* by a finite (perhaps empty) series of changes of bound variables, we shall say *P is congruent to Q*, or *P α-converts to Q*, or

$$P \equiv_\alpha Q$$

β-reduction

A term of form $(\lambda x.M)N$ represents an operator $\lambda x.M$ applied to an argument N [12, Page 11]. In the informal interpretation of $\lambda x.M$, its value when applied to N is calculated by substituting N for x in M. So $(\lambda x.M)N$ can be simplified to $[N/x]M$.

1.11 Definition (β-contracting, β-reduction)

Any term of form

$$(\lambda x.M)N$$

is called a *β-redex* and the corresponding term

$$[N/x]M$$

is called its *contractum*. A *contraction* occurs only when a term P containing an occurrence of $(\lambda x.M)N$ is replaced by $[N/x]M$ and the result is P'. We then say we have $contracted$ the redex-ocurrence in P, and P *β-contracts* to P' or

$$P \rhd_{1\beta} P',$$

If and only if P can be changed to Q by a finite (perhaps empty) series of β-contractions and changes of bound variables, we say *P β-reduces to Q*, or

$$P \rhd_{\beta} Q.$$

Examples:

(a) $\big(\lambda x.x(xy)\big)N \rhd_{1\beta} N(Ny)$

(b) $(\lambda x.y)N \rhd_{1\beta} y$

(c) $(\lambda x.(\lambda y.yx)z)v \rhd_{1\beta} [v/x]((\lambda y.yx)z) \equiv (\lambda y.yv)z \rhd_{1\beta} [z/y](yv) \equiv zv$

1.12 Definition

A term Q which contains no β-redexes is called a β-*normal form* (or a *term in* β-*normal form*) [12, Definition 1.26]. The class of all β-normal forms is called β-nf or $\lambda\beta$-nf. If a term P β-reduces to a term Q in β-nf, then Q is called a β-*normal form of* P.

1.13 Lemma

$$P \rhd_\beta Q \Longrightarrow \mathrm{FV}(P) \supseteq \mathrm{FV}(Q).$$

Proof can be found on [12, Lemma 1.30, Page 14].

1.14 Lemma (Substitution and $\rhd_\beta$)

If $P \rhd_\beta P'$ *and* $Q \rhd_\beta Q'$, *then*

$$[P/x]Q \rhd_\beta [P'/x]Q'.$$

Proof can be found on [12, Lemma 1.31, Page 14].

Example: We have

$$(\lambda x((\lambda y.y)u)x)z \equiv \underbrace{[((\lambda y.y)u)/w]}_{P}\underbrace{(\lambda x.wx)z}_{Q}$$

If $P' \equiv u$ and $Q' \equiv wz$ then $[P'/w]Q' \equiv uz$

Note that $(\lambda x(\lambda y.y)u)x)z \rhd uz$.

1.15 Theorem (Church-Rosser theorem for $\triangleright_\beta$)

If $P \triangleright_\beta M$ and $P \triangleright_\beta N$, then there exists a term T such that

$$M \triangleright_\beta T \text{ and } N \triangleright_\beta T.$$

Proof can be found on [12, Theorem 1.32, Page 14].

Example: $\underbrace{(\lambda x((\lambda y.\, y)u)x)z}_{P} \triangleright \underbrace{(\lambda x.\, ux)z}_{M} \triangleright uz$

$$\underbrace{(\lambda x((\lambda y.\, y)u)x)z}_{P} \triangleright \underbrace{((\lambda y.\, y)u)z}_{N} \triangleright \underbrace{uz}_{T}$$

***β*-equality**

We say P is β-equal or β-convertible to Q (notation $P =_\beta Q$) if and only if Q is obtained from P by a finite (perhaps empty) series of β-contractions and reversed β-contractions and changes of bound variables. That is, $P =_\beta Q$ if and only if there exists $P_0, \ldots, P_n (n \geq 0)$ such that

$$(\forall i \leq n-1)(P_i \triangleright_{1\beta} P_{i+1} \text{ or } P_{i+1} \triangleright_{1\beta} P_i \text{ or } P_i \equiv_\alpha P_{i+1}),$$

$$P_0 \equiv P, \qquad P_n \equiv Q.$$

1.16 Lemma

If $P =_\beta Q$ and $P \equiv_\alpha P'$ and $Q \equiv_\alpha Q'$, then $P' =_\beta Q'$.

[12, Lemma 1.39, Page 16].

Example: Same as in lemma 1.14.

1.17 Lemma (substitution lemma for β-equality)

$$M =_\beta M', N =_\beta N' \implies [N/x]M =_\beta [N'/x]M'$$

[12, Lemma 1.40, Page 16].

Example: Same as in theorem 1.15.

1.18 Theorem (Church-Rosser theorem for $=_\beta$)

If $P =_\beta Q$, then there exists a term T such that [4]

$$P \rhd_\beta T \text{ and } Q \rhd_\beta T$$

$\lambda\beta\eta$-reduction in λ-calculus

An η-redex is any λ-term

$$\lambda x.Mx$$

with $x \notin \mathrm{FV}(M)$. Its *contractum* is

$$M.$$

$\beta\eta$-reduction: A $\beta\eta$-redex is a β-redex or an η-redex. The phrases '*P $\beta\eta$-contracts to Q*' and '*P $\beta\eta$-reduces to Q*' are defined like '*β-contracts*' and '*β-reduces*' with notation

$$P \rhd_{1\beta\eta} Q, \qquad P \rhd_{\beta\eta} Q.$$

$\beta\eta$-normal form: A λ-term Q containing no $\beta\eta$-redexes is said to be in $\beta\eta$-normal form and we say such a term Q is a $\beta\eta$-normal form of P if and only if $P \rhd_{\beta\eta} Q$.

$\lambda\beta\eta$-equality in λ-calculus

We say P is $\beta\eta$-equal or $\beta\eta$-convertible to Q (notation $P =_{\beta\eta} Q$) if and only if Q is obtained from P by a finite (perhaps empty) series of β-contractions or η-contractions and reversed β-contractions or η-contractions and changes of bound variables. That is, $P =_{\beta\eta} Q$ if and only if there exists $P_0, \ldots, P_n (n > 0)$ such that

$$(\forall i \leq n-1)$$

$$(P_i \rhd_{1\beta\eta} P_{i+1} \quad \text{or} \quad P_{i+1} \rhd_{1\beta\eta} P_i \quad \text{or} \quad P_i \equiv_\alpha P_{i+1})$$

$$P_0 \equiv P, \qquad P_n \equiv Q.$$

$\lambda\beta$ and $\lambda\beta\eta$ theories

$\lambda\beta$ formal theory of β-equality

The formulas of $\lambda\beta$ are just equations $M = N$, for all λ-terms M and N. The axioms are the particular (α), (β) and (ρ) below, for all λ-terms M, N, and all variables x, y. The rules are (μ), (ν), (ξ), (τ), and (σ) below.

The axiom-schemes are:

(α) $\lambda x.M = \lambda y.[y/x]M$ if $y \notin \mathrm{FV}(M)$;

(β) $(\lambda x.M)N = [N/x]M$;

(ρ) $M = M$.

The rules of inference are:

(μ) $\frac{M=M`}{NM=NM`}$

(ν) $\frac{M=M`}{MN=M`N}$

(ξ) $\frac{M=M`}{\lambda x.M=\lambda x.M`}$

(τ) $\frac{M=N \quad N=P}{M=P}$

(σ) $\frac{M=N}{N=M}$

$\lambda\beta$ formal theory of β-reduction

This theory is called $\lambda\beta$ like the previous one. Its formulas are expressions $M \rhd N$, for all λ-terms M and N. Its axiom-schemes and rules are the same as above, but with '=' changed to '$\rhd$' and rule (σ) omitted. If and only if an expression $M \rhd N$ is provable in $\lambda\beta$, we say

$$\lambda\beta \vdash M \rhd N$$

The axiom-schemes are:

(α) $\lambda x.M \rhd \lambda y.[y/x]M$ if $y \notin \mathrm{FV}(M)$;

(β) $(\lambda x.M)N \rhd [N/x]M$;

(ρ) $M \rhd M$.

The rules of inference are:

(μ) $\frac{M \rhd M`}{NM \rhd NM`}$

(ν) $\frac{M \rhd M`}{MN \rhd M`N}$

(ξ) $\frac{M \rhd M`}{\lambda x.M \rhd \lambda x.M`}$

(τ) $\frac{M \rhd N \quad N \rhd P}{M \rhd P}$

1.19 Lemma

(a) $M \rhd_\beta N \Leftrightarrow \lambda\beta \vdash M \rhd N$;

(b) $M =_\beta N \Leftrightarrow \lambda\beta \vdash M = N$.

Proof can be found on [12, Lemma 6.4, Page 71].

***λβη* formal theory of *βη*-equality**

Consider the rules of inference:

(ζ) $\frac{Mx = Nx}{M = N}$ if $x \notin \mathrm{FV}(MN)$

(η) $\lambda x.Mx = M$ if $x \notin \mathrm{FV}(M)$.

Let $\lambda\beta$ be the theory of equality as defined above. We define two new theories of equality:

$\lambda\beta\zeta$: add rule (ζ) to $\lambda\beta$

$\lambda\beta\eta$: add axiom-scheme (η) to $\lambda\beta$

(Adding (η) means adding all equations $\lambda x.Mx = M$ as new axioms, for all terms M and all $x \notin \mathrm{FV}(M)$).

$\lambda\beta\eta$ formal theory of $\beta\eta$-reduction

This is defined by adding to the theory of β-reduction (discussed above), the axiom scheme

(η) $\lambda x.Mx \rhd M$ (if $x \notin \mathrm{FV}(M)$)

Thus the axiom schemes are:

(α) $\lambda x.M \rhd \lambda y.[y/x]M$ if $y \notin \mathrm{FV}(M)$;

(β) $(\lambda x.M)N \rhd [N/x]M$;

(ρ) $M \rhd M$;

(η) $\lambda x.Mx \rhd M$ (if $x \notin \mathrm{FV}(M)$).

The rules of inference are:

(μ) $\frac{M \rhd M`}{NM \rhd NM`}$

(ν) $\frac{M \rhd M`}{MN \rhd M`N}$

(ξ) $\frac{M \rhd M`}{\lambda x.M \rhd \lambda x.M`}$

$$(\tau)\qquad \frac{M \rhd N \quad N \rhd P}{M \rhd P}$$

The theories $\lambda\beta\zeta$ and $\lambda\beta\eta$ of equality are equivalent. Thus both theories determine the same equality relation.

1.20 Lemma

Rule (ζ) *is equivalent to the combination of rule* (ξ) *and rule* (η).

$$(\xi) + (\eta) \Longrightarrow (\zeta)$$

$$x \notin \mathrm{FV}(MN)$$

$$\frac{\dfrac{\dfrac{}{M = \lambda x.Mx}\ (\eta) \quad \dfrac{Mx = Nx}{\lambda x.Mx = \lambda x.Nx}\ (\xi)}{M = \lambda x.Nx}\ (\tau) \quad \dfrac{}{\lambda x.Nx = N}\ (\eta)}{M = N}\ (\tau)$$

$$(\zeta) \Longrightarrow (\xi)$$

$$\frac{\dfrac{\dfrac{}{(\lambda x.M)x = M}\ (\beta) \quad M = N}{(\lambda x.M)x = N}\ (\tau) \quad \dfrac{}{N = (\lambda x.N)x}\ (\beta)}{\dfrac{(\lambda x.M)x = (\lambda x.N)x}{\lambda x.M = \lambda x.N}\ (\zeta)}\ (\tau)$$

$$(\zeta) \Longrightarrow (\eta)$$

$$x \notin \mathrm{FV}(M)$$

$$\frac{\dfrac{}{(\lambda x.Mx)x = Mx}\ (\beta)}{\lambda x.Mx = M}\ (\zeta)$$

Chapter 2

Combinatory Logic

Systems of combinators are designed to perform the same tasks as systems of λ-calculus, but without using bound variables [12, Chapter 2]. To motivate combinators, consider the commutative law of addition in arithmetic, which says

$$(\forall x, y)(x + y) = (y + x)$$

The above expression contains bound variables '*x*' and '*y*'. But these can be removed, as follows. We first define an addition operator *A* by

$$A(x, y) = x + y \qquad \text{(for all } x, y\text{)},$$

and then introduce an operator **C** defined by

$$\big(\mathbf{C}(f)\big)(x, y) = f(y, x) \qquad \text{(for all } f, x, y\text{)}.$$

Then the commutative law becomes simply

$$A = \mathbf{C}(A).$$

The operator **C** may be called a *combinator*; the other examples of such an operator are the following:

B, which composes two functions: $\big(\mathbf{B}(f, g)\big)(x) = f(g(x))$;

$\mathbf{B}'$, a reversed composition operator: $\big(\mathbf{B}'(f, g)\big)(x) = g(f(x))$;

I, the identity operator: $\mathbf{I}(f) = f$;

K, which forms constant functions: $\big(\mathbf{K}(a)\big)(x) = a$;

S, a stronger composition operator: $\big(\mathbf{S}(f,g)\big)(x) = f(x, g(x))$;

W, for doubling: $\big(\mathbf{W}(f)\big)(x) = f(x,x)$.

In this first section, same notation as in [12, Chapter 2] has been used.

2.1 Definition (Combinatory Logic terms, or CL-terms)

Assume that there is an infinite sequence of expressions $v_{0,}v_{00}, v_{000}, \ldots$ called *variables* [12, Definition 2.1], and a finite or infinite sequence of expressions called *atomic constants*, including three called basic *combinators*: **I**, **K**, **S**. (If **I**, **K** and **S** are the only atomic constants, the system will be called *pure*, otherwise *applied.*) The set of expressions called *CL-terms* is defined inductively as follows:

(a) All variables and atomic constants, including **I**, **K**, **S**, are CL-terms.

(b) If *X* and *Y* are CL-terms, then so is (*XY*).

An *atom* is a variable or atomic constant. A *non-redex constant* is an atomic constant other than **I**, **K**, **S**. A *non-redex atom* is a variable or a non-redex constant. A *closed term* is a term containing no variables. A *combinator* is a term whose only atoms are basic combinators. (In the pure system this is the same as a closed term.)

Examples of CL-terms**:** ((**S**(**KS**))**K**), ((**S**(**K** x))((**SK**)**K**)).

2.2 Definition (Length of a term)

The *length of X* (or *lgh*(*X*)) is the number of occurrences of atoms in *X* [12, Definition 2.3]:

(a) *lgh(a)* = 1 for atoms *a*;

(b) *lgh(UV) = lgh(U) + lgh(V)*.

2.3 Definition (Occurrence of a variable)

The relation *X occurs in Y*, or *X is a subterm of Y* [12, Definition 2.4], is defined thus:

(a) *X* occurs in *X*;

(b) If *X* occurs in *U* or in *V*, then *X* occurs in (*UV*).

The set of all variables occurring in *Y* is called FV(*Y*).

2.4 Definition (Substitution)

$[U/x]Y$ is defined to be the result of substuting *U* for every occurrence of x in *Y* [12, Definition 2.6]; that is,

(a) $[U/x]x \equiv U$,

(b) $[U/x]a \equiv a$ for atoms $a \not\equiv x$,

(c) $[U/x](VW) \equiv (([U/x]V)([U/x]W))$.

For all $U_1, \dots, U_n$ and mutually distinct $x_1, \dots, x_n$, the result of simultaneously substuting U_1 for x_1, U_2 for x_2, …, U_n for x_n in Y is called

$$[U_1/x_1\ , \dots , U_n/x_n\ ]Y.$$

2.5 Definition (Weak Reduction)

Any term **I***X*, **K***XY* or **S***XYZ* is called a (*weak*) *redex* [12, Definition 2.9]. *Contracting* an occurrence of a weak redex in a term *U* means replacing one occurrence of

I*X* by *X*, or

K*XY* by *X*, or

S*XYZ* by *XZ(YZ)*.

If and only if this changes *U* to U', we say that *U (weakly) contracts to* U', or

$$U \rhd_{1w} U'.$$

If and only if *V* is obtained from *U* by a finite (perhaps empty) series of weak contractions, we say that *U (weakly) reduces to V*, or

$$U \rhd_{w} V.$$

2.6 Definition

A *weak normal form* (or *weak nf* or *term in weak normal form*) is a term containing no weak redexes [12, Definition 2.10]. If *U* weakly reduces to a weak normal form *X*, then *X* is called a *weak normal form of U*.

Example: Define **B** ≡ **S**(**KS**)**K**. Then **B***XYZ* $\rhd_w$ $X(YZ)$, since

$$\mathbf{B}XYZ \equiv \mathbf{S}(\mathbf{KS})\boldsymbol{K}XYZ$$

$\triangleright_{1w}$ **KS***X*(**K***X*)*YZ* by contracting **S**(**KS**)**K***X* to **KS***X*(**K***X*)

$\triangleright_{1w}$ **S**(**K***X*)*YZ* by contracting **KS***X* to **S**

$\triangleright_{1w}$ **K***XZ*(*YZ*) by contracting **S**(**K***X*)*YZ*

$\triangleright_{1w}$ *X*(*YZ*) by contracting **K***XZ*

Here also the parentheses are left associative.

Example:

SK*xy* ≡ ((**SK**)*x*)*y*

SIK(*xy*) ≡ ((**SI**)**K**)(*xy*)

2.7 Definition (Abstraction)

For every CL-term M and every variable x [12, Definition 2.18], a CL-term called $[x].M$ is defined by induction on M, thus:

(a) $[x].M \equiv \mathbf{K}M$ if $x \notin$FV (M);

(b) $[x].x \equiv \mathbf{I}$;

(c) $[x].Ux \equiv U$ if $x \notin$FV (U);

(d) $[x].UV \equiv \mathbf{S}([x].U)([x].V)$ if neither (a) nor (c) applies

Example: $[x].xy \equiv \mathbf{S}([x].x)([x].y)$

$\equiv \mathbf{SI}(\mathbf{K}y)$

2.8 Theorem

The clauses in definition 2.7 allow us to construct $[x].M$ *for all* x *and* M [12, Theorem 2.21]. *Further,* $[x].M$ *does not contain* x*, and for all* N*,*

$$([x].M)N \rhd_w [N/x]M.$$

Proof: By structural induction on the construction of M, the proof has the following cases:

Case 1: $x \notin \mathrm{FV}(M)$. Then

$$[x].M \equiv \mathbf{K}M.$$

Since $x \notin \mathrm{FV}(M)$, $x \notin \mathrm{FV}(\mathbf{K}M) = \mathrm{FV}([x].M)$.

Also $$([x].M)N \equiv \mathbf{K}MN \rhd M \equiv [N/x]M.$$

Case 2: $M \equiv x$. Then

$$[x].M \equiv [x].x \equiv \mathbf{I}.$$

Here $x \notin \mathrm{FV}([x].x) \equiv FV(\mathbf{I})$ and,

$$([x].x)N \equiv \mathbf{I}N \rhd N \equiv [N/x]x.$$

Case 3: $M \equiv Ux$, where $x \notin \mathrm{FV}(U)$.

Then, $[x].M \equiv U$ and $x \notin \mathrm{FV}(U) = \mathrm{FV}([x].Ux)$.

Also, $$([x].Ux)N \equiv UN \equiv [N/x].Ux.$$

Case 4: $M \equiv UV$ and neither of cases 1 or 3 applies. Then,

$$[x].M \equiv \mathbf{S}([x].U)([x].V).$$

Also, by the induction hypothesis,

$x \notin \mathrm{FV}([x].U)$ and $([x].U)N \rhd [N/x]U$ and also $x \notin \mathrm{FV}([x].V)$ and $([x].V)N \rhd [N/x]V$.

Then, $x \notin \mathrm{FV}(\mathbf{S}([x].U)([x].V))$ and

$$([x].M)N \equiv \mathbf{S}([x].U)([x].V)N$$

$$\rhd ([x].U)N([x].V)N$$

$$\rhd [N/x]U([N/x]V)$$

$$\rhd [N/x](UV) \equiv [N/x]M.$$

Example:

(a) $[x, y].x \equiv [x].([y].x)$

$\equiv [x].\mathbf{K}x$

$\equiv \mathbf{K}$

(b) $[x, y, z].xz(yz) \equiv [x].([y].([z].xz(yz)\,)\,)$

$\equiv [x].([y].(\mathbf{S}([z].xz)([z].yz)\,)\,)$

$\equiv [x].([y].\mathbf{S}xy)$

$\equiv [x].\mathbf{S}x$

$\equiv \mathbf{S}$

Formal theory of weak equality

The formulas of CL*w* are equations $X = Y$, for all CL-terms X and Y [12, Definition 6.5]. The axioms are the particular cases of the four axiom-schemes below, for all CL-terms X, Y and Z. The rules are (μ), (ν), (τ) and (σ) below.

The axiom-schemes are:

(**I**) $\mathbf{I}X = X$;

(**K**) $\mathbf{K}XY = X$;

(**S**) $\mathbf{S}XYZ = XZ(YZ)$;

(ρ) $X = X$.

The rules of inference are:

(μ) $\dfrac{X=X`}{ZX=ZX`}$

(ν) $\dfrac{X=X`}{XZ=X`Z}$

(τ) $\dfrac{X=Y \quad Y=Z}{X=Z}$

(σ) $\dfrac{X=Y}{Y=X}$

If and only if an equation $X = Y$ is provable in CL*w*, we say

$$\mathrm{CL}w \vdash X = Y.$$

Formal theory of weak reduction

The formulas of CL*w* are expressions $X \rhd Y$, for all CL-terms *X* and *Y* [12, Definition 6.6]. The axiom-schemes and rules are the same as above [10], but with '=' changed to '$\rhd$' and (σ) omitted. If and only if $X \rhd Y$ is provable in CL*w*, we say

$$CLw \vdash X \rhd Y.$$

Thus the axiom schemes are:

(**I**) **I***X* $\rhd$ *X*;

(**K**) **K***XY* $\rhd$ *X*;

(**S**) **S***XYZ* $\rhd$ *XZ*(*YZ*);

(ρ) $X \rhd X$.

The rules of inference are:

(μ) $\frac{X \rhd X`}{ZX \rhd ZX`}$

(ν) $\frac{X \rhd X`}{XZ \rhd X`Z}$

(τ) $\frac{X \rhd Y \quad Y \rhd Z}{X \rhd Z}$

2.9 Lemma

(a) $X \rhd_w Y \Leftrightarrow \mathrm{CL}w \vdash X \rhd Y$

(b) $X =_w Y \Leftrightarrow \mathrm{CL}w \vdash X = Y$

[12, Lemma 6.7, Page 71].

Chapter 3

Correspondence Between λ-calculus and Combinatory Logic

Even though the terms in lambda calculus (λ-claculus) and combinatory logic (CL) appear to be equivalent, they are not. In order to understand this, we must first look at the λ-transform and the H-transform.

3.1 λ-Transform

To each CL-term X we associate a λ-term X_λ called its λ-transform, by induction on X [12, Definition 9.2]. Thus:

(a) $x_\lambda \equiv x$

(b) $\mathbf{I}_\lambda \equiv \lambda x.x, \qquad \mathbf{K}_\lambda \equiv \lambda xy.x, \qquad \mathbf{S}_\lambda \equiv \lambda xyz.xz(yz)$

(c) $(XY)_\lambda \equiv X_\lambda Y_\lambda$

3.2 H-Transform

To each λ-term M we associate a CL-term called M_H [12, Definition 9.10]. Thus:

(a) $x_H \equiv x$

(b) $(MN)_H \equiv M_H N_H$

(c) $(\lambda x.M)_H \equiv [x].(M_H)$

The λ-transform and the H-transform allow us to describe λ-terms in CL and vice versa.

3.3 Weak normal form

A weak normal form (or weak nf or term in weak normal form) is a term that contains no weak redexes.

For example, in combinatory logic;

$$[x].(\mathbf{K}xx) \equiv \mathbf{S}([x].\mathbf{K}x)([x].x) \equiv \mathbf{SKI}$$

The above is in weak normal form [1, Page 152].

If we look at the equivalent term in λ-calculus, we get;

$$\lambda x.\mathbf{K}_\lambda xx \equiv \lambda x.(\lambda uv.u)xx \rhd_\beta \lambda x.x \equiv \mathbf{I}_\lambda$$

Here, $\lambda x.\mathbf{K}_\lambda xx$ does reduce to $\mathbf{I}_\lambda$.

The key point is that [*x*].(**K***xx*) ≡ **SKI** cannot be reduced in CL, although **K***xx* can be reduced, because **K***xx* really does not occur as a subterm of [*x*].(**K***xx*) ≡ **SKI**, whereas $\mathbf{K}_\lambda xx$ really does occur as a subterm of $\lambda x.\mathbf{K}_\lambda xx$. So the latter term reduces to $\mathbf{I}_\lambda$.

We have seen that λ-calculus and CL are different, but there are some points of similarity. We have the combinatory β-equality ($=_{c\beta}$) and combinatory $\beta\eta$-equality ($=_{c\beta\eta}$), which are equivalent to $\lambda\beta$-conversion and $\lambda\beta\eta$-conversion, respectively. To understand this, let us first take a brief overview of extensional equality in CL.

3.4 Extensional Equality in CL

Consider the rules:

(ζ) $\quad \frac{Xx=Yx}{X=Y}$ $\qquad$ if $x \notin \mathrm{FV}(XY)$

(ξ) $\dfrac{X=Y}{[x].X=[x].Y}$

Consider the axiom-scheme:

(η) $[x].Ux = U$ if $x \notin \mathrm{FV}(U)$

Here, the relation $=_{ext}$ is defined as:

$$X =_{ext} Y \;\Leftrightarrow\; \mathrm{CL}\zeta \vdash X = Y$$

Where, CLζ can be obtained by adding the rule (ζ) to CL*w* (formal theory of weak equality defined in Chapter 2).

Thus, the axiom-schemes are:

(**I**) **I***X* = *X*;

(**K**) **K***XY* = *X*;

(**S**) **S***XYZ* = *XZ*(*YZ*);

(ρ) *X* = *X*.

The rules of inference are:

(μ) $\dfrac{X=X`}{ZX=ZX`}$

(ν) $\dfrac{X=X`}{XZ=X`Z}$

(τ) $\dfrac{X=Y \quad Y=Z}{X=Z}$

(σ) $\dfrac{X=Y}{Y=X}$

(ζ) $\frac{Xx=Yx}{X=Y}$

This relation is often called $=_{c\beta\eta}$.

Example: **SK** $=_{ext}$ **KI**

This is proved by applying rule (ζ) twice to the weak equation **SK**$xy =_w$ **KI**xy, which is proved thus;

SK$xy =_w$ **K**$y(xy)$

$=_w y$

$=_w$ **I**y

$=_w$ **KI**xy

3.5 Extensionality axioms

The theory $\mathrm{CL}_{ext\,ax}$ (i.e. $=_{ax}$) is defined by adding to CLw the following five axioms [12, Definition 8.10]:

E-ax 1: **S**(**S**(**KS**)(**S**(**KK**)(**S**(**KS**)**K**)))(**KK**) = **S**(**KK**);

E-ax 2: **S**(**S**(**KS**)**K**)(**KI**) = **I**;

E-ax 3: **S**(**KI**) = **I**;

E-ax 4: **S**(**KS**)(**S**(**KK**)) = **K**;

E-ax 5: **S**(**K**(**S**(**KS**)))(**S**(**KS**)(**S**(**KS**))) = **S**(**S**(**KS**)(**S**(**KK**)(**S**(**KS**)(**S**(**K**(**S**(**KS**)))**S**))))(**KS**).

Since the theory $\mathrm{CL}_{ext\,ax}$ is equivalent to the $\mathrm{CL}_{\beta\eta}$ theory, it determines the same equality-relation, namely $=_{ext}$.

Now let us discuss about the relation of β-equality in λ-calculus.

3.6 Combinatory β-equality

The relation of β-equality in λ-calculus induces the following equality between CL-terms [12, Definition 9.29].

For all CL-terms *X* and *Y*, define

$$X =_{c\beta} Y \Leftrightarrow X_\lambda =_\beta Y_\lambda$$

3.7 Functional CL-terms

A CL-term with one of the six forms **S***XY* (for some *X*, *Y*), **S***X*, **K***X*, **S**, **K**, **I**, is called functional or fnl [12, Definition 9.6].

3.8 Lemma

For all functional CL-terms *U*:

(a) $U_\lambda \rhd_\beta \lambda x.M$ for some λ-term *M*;

(b) $U \rhd_w V \Longrightarrow V$ is functional.

[12, Lemma 9.7, Page 94].

3.9 The formal theory CLζ_{β}

CLζ_{β} is obtained by adding the following rule to the theory CL*w* of weak equality [12, Definition 9.32]:

(ζ_{β}) $\frac{Ux=Vx}{U=V}$ if $x \notin \mathrm{FV}(UV)$ and U and V are functional.

The axiom schemes become:

(**I**) **I***X* = *X*;

(**K**) **K***XY* = *X*;

(**S**) **S***XYZ* = *XZ*(*YZ*);

(ρ) *X* = *X*.

The rules of inference are:

(μ) $\frac{X=X`}{ZX=ZX`}$

(ν) $\frac{X=X`}{XZ=X`Z}$

(τ) $\frac{X=Y \quad Y=Z}{X=Z}$

(ζ_{β}) $\frac{Ux=Vx}{U=V}$ if $x \notin \mathrm{FV}(UV)$ and U and V are functional.

Example: CLζ_{β} ⊢ **SK** = **KI**.

$$\mathbf{SK}yz = \mathbf{K}z(yz) = z = \mathbf{I}z.$$

The rule (ζ_β) can be applied, since **SK**y and **I** are functional, to give

$$\mathbf{SK}y = \mathbf{I}.$$

But CL$w \vdash \mathbf{I} = \mathbf{KI}y$, so CL$\zeta_\beta \vdash \mathbf{SK}y = \mathbf{KI}y$. Since **SK** and **KI** are functional, rule (ζ_β) can be applied to give

$$\mathbf{SK} = \mathbf{KI}.$$

The above holds true for conversion, i.e., describing the conversion in λ-calculus and trying to find its equivalent in combinatory logic, but what about reduction? It so happens that we have Strong Reduction in CL which is equivalent to $\lambda\beta\eta$-reduction in λ-calculus. This is the subject of the next section.

3.10 Definition (Strong reduction, $\succ$)

The formal theory of *strong reduction* has as formulas all expressions $X \succ Y$ [12, Definition 8.15], for all CL-terms X and Y. Its *axiom-schemes* and *rules* are the same as those for CLw, but with '=' changed to '$\succ$', and the following new rule added:

(ξ) $\qquad \dfrac{X \succ Y}{[x].X \succ [x].Y}$

Thus, the axiom-schemes become:

(**I**) $\quad \mathbf{I}X \succ X$;

(**K**) $\quad \mathbf{K}XY \succ X$;

(**S**) $\quad \mathbf{S}XYZ \succ XZ(YZ)$;

(ρ) $X \succ\!\!- X.$

The rules of inference are:

(μ) $\dfrac{X \succ\!\!- X`}{ZX \succ\!\!- ZX`}$

(ν) $\dfrac{X \succ\!\!- X`}{XZ \succ\!\!- X`Z}$

(τ) $\dfrac{X \succ\!\!- Y \quad Y \succ\!\!- Z}{X \succ\!\!- Z}$

(ξ) $\dfrac{X \succ\!\!- Y}{[x].X \succ\!\!- [x].Y}$

Also, rule (ξ) can be replaced with rule (ξ`) which says that;

(ξ`) $\dfrac{Ux \succ\!\!- Y}{U \succ\!\!- [x]Y} \qquad (x \notin \mathrm{FV}(U))$

(ξ`) can be derived from (ξ) by taking $[x].X$ for $U$, and (ξ`) from (ξ) by taking Ux for X. The U in (ξ`) can be restricted to being a functional term [6, Page 93].

If and only if $X \succ\!\!- Y$ is provable in this theory, we say *X strongly reduces to Y*, or just

$X \succ\!\!- Y$.

The strong reduction was proposed by H.B. Curry. At that time, he used a formalism that included both the abstraction operator λ as primitive and also included the basic combinators **I**, **K**, and **S**, so that abstraction could be defined by the definition of $[\ ]^{\eta}$ [Definition 2.7, Page 24]. So, if we were to perform strong reduction to prove that **S(KI)** $\succ\!\!-$ **I**, we would write it in the following way;

S(**KI**)

$$\lambda x.\,\mathbf{S}(\mathbf{KI})x$$

$$\lambda x.\,\lambda y.\,\mathbf{S}(\mathbf{KI})xy$$

$$\lambda xy.\,\mathbf{S}(\mathbf{KI})xy$$

$$\lambda xy.\,\mathbf{KI}y(xy)$$

$$\lambda xy.\,\mathbf{I}(xy)$$

$$\lambda xy.\,xy$$

$$\lambda x.\,x$$

$$\mathbf{I}$$

But later, he used the standard syntax for CL in which λx was replaced with $[x]$. So, the reduction would then be;

$$\mathbf{S}(\mathbf{KI})$$

$$[x].\,\mathbf{S}(\mathbf{KI})x$$

$$[x].\,[y].\,\mathbf{S}(\mathbf{KI})xy$$

$$[x].\,[y].\,\mathbf{S}(\mathbf{KI})xy$$

$$[x].\,[y].\,\mathbf{KI}y(xy)$$

$$[x].\,[y].\,\mathbf{I}(xy)$$

$$[x].\,[y].\,xy$$

$[x].x$

I

Examples:

(a) SK $\succ$ KI

To prove this, first note that $\mathbf{SK}xy \rhd_w \mathbf{K}y(xy) \rhd_w y$. Since the axiom-schemes and rules for $\succ$ include those for $\rhd_w$, this gives

$\mathbf{SK}xy \succ y$.

Hence, by rule (ξ) twice,

$[x,y].\mathbf{SK}xy \succ [x,y].y$

But $[x,y].\mathbf{SK}xy \equiv \mathbf{SK}$ and $[x,y].y \equiv \mathbf{KI}$.

(b) S(K*X*)(K*Y*) $\succ$ K(*XY*)

To prove this for all terms *X* and *Y*, choose $v \notin \mathrm{FV}(XY)$. Then

$\mathbf{S}(\mathbf{K}X)(\mathbf{K}Y) \equiv [v].(\mathbf{K}Xv)(\mathbf{K}Yv)$, $\mathbf{K}(XY) \equiv [v].XY$

Also $(\mathbf{K}Xv)(\mathbf{K}Yv) \rhd_w XY$, so by ($\xi$),

$[v].(\mathbf{K}Xv)(\mathbf{K}Yv) \succ [v].XY$

3.11 Lemma

The relation $\succ\!\!-$ *is transitive and reflexive. Also*

(a) $X \succ\!\!- Y \Rightarrow \mathrm{FV}(X) \supseteq \mathrm{FV}(Y)$;

(b) $X \succ\!\!- Y \Rightarrow [X/v]Z \ \succ\!\!- [Y/v]Z$;

(c) $X \succ\!\!- Y \Longrightarrow [U_1/x_1 , \ldots . , U_n/x_n \,]X \succ\!\!- [U_1/x_1 , \ldots . , U_n/x_n \,]Y$;

(d) *the equivalence relation generated by* $\succ\!\!-$ *is the same as* $=_{ext}$; *that is,* $X =_{ext} Y$ *if and only if X goes to Y by a finite series of strong reductions and reversed strong reductions.*

Proof can be found on [12, Lemma 8.17, Page 90].

The theory of strong reduction depends on the definition of the abstraction being used. In particular, it requires an abstraction using clause (c) of the abstraction algorithm, which is:

$$[x].Ux \equiv U \text{ if } x \notin \mathrm{FV}(U)$$

This definition does not work for defining a reduction in combinatory logic equivalent to $\lambda\beta$-reduction. This is because rule (η) is not valid for all $\lambda\beta$-reductions but it is valid for some. So some instances of clause (c) are needed but not all.

If $x \notin \mathrm{FV}(V)$ and $x \not\equiv y$, we have

$$\lambda x.(\lambda y.V)x \rhd_\beta \lambda x.[x/y]V \rhd_\alpha \lambda y.V$$

But clause (c) of the abstraction cannot be restricted to combinatory logic terms equivalent to abstracts because there is no algorithm which can decide whether a CL term is equivalent to an abstract, and so the result is not an algorithm.

We are now going to look at various alternative definitions of abstractions, as the discussion above shows the need for this.

So far, we have seen [ ] . From now on, [ ] will be called $[\ ]^{\eta}$. Now let us look at $[\ ]^{w}$. Here we must note that *H* is really H_{η}.

Weak Abstraction

For all CL-terms Y, $[x]^{w}.Y$ is defined thus:

(a) $[x]^{w}.Y \equiv \mathbf{K}Y$ if $x \notin \mathrm{FV}(Y)$;

(b) $[x]^{w}.x \equiv \mathbf{I}$;

(f) $[x]^{w}.UV \equiv \mathbf{S}([x]^{w}.U)([x]^{w}.V)$ if $x \in \mathrm{FV}(UV)$.

3.12 Lemma

For all CL-terms Y and Z, $[x]^{w}.Y$ *is defined for all variables x and terms Y and does not contain x.*

3.13 Lemma

$$([x]^{w}.Y)Z \rhd_{w} [Z/x]Y$$

Proof:

Case 1: $x \notin \mathrm{FV}(Y)$

Then, $([x]^w.Y)Z \equiv \mathbf{K}YZ \rhd_w Y \equiv [Z/x]Y$

Case 2: $Y \equiv x$

Then, $([x]^w.Y)Z \equiv ([x]^w.x)Z \equiv \mathbf{I}Z \rhd_x Z \equiv [Z/x]x$

Case 3: $Y \equiv Y_1Y_2$ and $x \in \mathrm{FV}(Y_1Y_2)$

Then, by the induction hypothesis, $([x]^w.Y_1)Z \rhd_w [Z/x]Y_1$ and $([x]^w.Y_2)Z \rhd_w [Z/x]Y_2$.

$$([x]^w.Y)Z \equiv ([x]^w.(Y_1Y_2))Z$$

$$\equiv \mathbf{S}([x]^w.Y_1)([x]^w.Y_2)Z$$

$$\rhd_w ([x]^w.Y_1)Z(([x]^w.Y_2)Z)$$

$$\rhd_w [Z/x]Y_1([Z/x]Y_2)$$

$$\equiv [Z/x](Y_1Y_2)$$

$$\equiv [Z/x]Y$$

3.14 Lemma

$$[z]^w.[z/x]Y \equiv [x]^w.Y \qquad \text{if } z \notin \mathrm{FV}(Y)$$

Proof:

Case 1: $x \notin \mathrm{FV}(Y)$

Then, $[x]^w.Y \equiv \mathbf{K}Y$

$$[z]^w.[z/x]Y \equiv [z]^w.Y \equiv \mathbf{K}Y \equiv [x]^w.Y$$

Case 2: $Y \equiv x$

Then, $[z]^w.[z/x]Y \equiv [z]^w.[z/x]x \equiv [z]^w.z \equiv \mathbf{I} \equiv [x]^w.x \equiv [x]^w.Y$

Case 3: $Y = Y_1Y_2$

Then, by the induction hypothesis, $[z]^w[z/x]Y_1 \equiv [x]^wY_1$ and $[z]^w[z/x]Y_2 \equiv [x]^wY_2$.

$$[z]^w[z/x]Y \equiv [z]^w[z/x]Y_1Y_2$$

$$\equiv \mathbf{S}([z]^w.[z/x]Y_1)([z]^w.[z/x]Y_2)$$

$$\equiv \mathbf{S}([x]^w.Y_2)([x]^w.Y_2)$$

$$\equiv [x]^w.Y$$

3.15 Lemma

$$[Z/v]([x]^w.Y) \equiv [x]^w.([Z/v]Y) \qquad \text{if } x \notin \mathrm{FV}(vZ)$$

Proof:

Case 1: $x \notin \mathrm{FV}(Y)$

Then $[Z/v]([x]^w.Y) \equiv [Z/v].\mathbf{K}Y \equiv \mathbf{K}Y \equiv \mathbf{K}.([Z/v]Y) \equiv [x]^w.[Z/v]Y.$

Case 2: $Y \equiv x$

Then, $[Z/v]([x]^w.x) \equiv [Z/v]\mathbf{I} \equiv \mathbf{I} \equiv [x]^w.x.$

Case 3: $Y = Y_1Y_2$

Then, by the induction hypothesis,

$[Z/v]([x]^w.Y_1) \equiv [x]^w.[Z/v]Y_1$ and $[Z/v]([x]^w.Y_2) \equiv [x]^w.[Z/v]Y_2$.

$$[Z/v]([x]^w.Y_1Y_2)$$

$$\rhd_w\ [Z/v]\mathbf{S}([x]^w.Y_1)([x]^wY_2)$$

$$\equiv S([Z/v]([x]^w.Y))([Z/v]([x]^w.Y))$$

$$\equiv \mathbf{S}([x]^w.[Z/v]Y_1)([x]^w.[Z/v]Y_2)$$

$$\equiv [x]^w.[Z/v]Y$$

3.16 Lemma

$$([x]^w.Y)_\lambda =_\beta \lambda x.(Y_\lambda)$$

Proof:

Case 1: $x \notin \mathrm{FV}(Y)$

$$[x]^w.Y \equiv \mathbf{K}Y$$

$$([x]^w.Y)_\lambda \equiv (\mathbf{K}Y)_\lambda$$

$$\equiv (\lambda uv.u)(Y_\lambda) \qquad \text{where } uv \notin \mathrm{FV}(Y) = \mathrm{FV}(Y_\lambda)$$

$$\rhd_\beta \lambda v.(Y_\lambda) \qquad \text{where } v \notin \mathrm{FV}(Y)$$

$$\equiv_\alpha \lambda x.(Y_\lambda)$$

Case 2: $Y \equiv x$

Then, $([x]^w.x)_\lambda \equiv \mathbf{I}_\lambda \equiv (\lambda x.x) \equiv \lambda x.(Y_\lambda)$

Case 3: $Y = Y_1 Y_2$

Then, by the induction hypothesis, $([x]^w.Y_1)_\lambda = \lambda x.Y_{1_\lambda}$ and $([x]^w.Y_2)_\lambda = \lambda x.Y_{2_\lambda}$.

$$([x]^w.Y)_\lambda \equiv ([x]^w.Y_1Y_2)_\lambda \equiv (\mathbf{S}([x]^w.Y_1)([x]^w.Y_2))_\lambda$$

$$\equiv_\beta (\lambda uvw.uw(vw))(\lambda x.Y_{1_\lambda})(\lambda x.Y_{2_\lambda})$$

$$\equiv_\beta \lambda w.(\lambda x.Y_{1_\lambda})w((\lambda x.Y_{2_\lambda})w)$$

$$\equiv_\beta \lambda w.([w/x].Y_{1_\lambda})([w/x].Y_{2_\lambda})$$

$$\equiv_\alpha \lambda x.Y_{1_\lambda}Y_{2_\lambda}$$

$$\equiv \lambda x.Y_\lambda$$

Abstraction $[\ ]^\beta$

Curry defined an abstraction, today called '$[\ ]^\beta$', which does not admit clause (c) of Definition 2.7 in all cases [12, Remark 9.27]. His definition consists of weak abstraction clauses (a) and (b), plus the following:

(c_β) $[x]^\beta.Ux \equiv U$ if U is functional and $x \notin \mathrm{FV}(U)$;

(f_β) $[x]^\beta.UV \equiv S([x]^\eta.U)([x]^\eta.V)$ if neither (a) nor (c_β) applies.

Thus this abstraction is defined by:

(a) $[x]^{\beta}.M \equiv \mathbf{K}M$ if $x \notin$FV (M);

(b) $[x]^{\beta}.x \equiv \mathbf{I}$;

(c_{β}) $[x]^{\beta}.Ux \equiv U$ if U is functional and $x \notin$ FV(U);

(f_{β}) $[x]^{\beta}.UV \equiv S([x]^{\eta}.U)([x]^{\eta}.V)$ if neither (a) nor (c_{β}) applies.

Note the two η's in (f_{β}); their effect is to say that clause (c) can be used unrestrictedly in computing $[x].Y$ if it is not the first clause in the evaluation.

The H_{β} mapping

For all λ-terms M, define H_{β} as in the definition of H mapping, but using $[\ ]^{\beta}$ instead of [ ]:

$$(\lambda x.M)_{H_{\beta}} \equiv [x]^{\beta}(M_H)$$

The H_w mapping

For all λ-terms M, we define M_{H_w}, but using $[\ ]^{w}$ instead of $[\ ]^{\eta}$; in particular, define

$$(\lambda x.M)_{H_w} \equiv [x]^{w}.(M_{H_w})$$

For all λ-terms M and N:

(a) $\mathrm{FV}(M_{H_w}) = \mathrm{FV}(M)$;

(b) $M \equiv_\alpha N \Longrightarrow M_{H_w} \equiv N_{H_w}$;

(c) $([N/x]M)_{H_w} \equiv [N_{H_w}/x](M_{H_w})$.

So, we have seen that we have strong reduction in CL which is equivalent to $\lambda\beta\eta$-reduction in λ-calculus. But then what about $\lambda\beta$-reduction (in λ-calculus)? As of now, we do not have complete equivalent reduction in CL and it is the only part of CL that is now missing. However, there are a few proposals by Curry, Seldin and Mezghiche which are discussed below.

Notations:

The (ξ)-rule is called (ξ_β) or (ξ_η), depending on the type of abstraction being used.

Curry's restriction to clause (c)

H.B. Curry proposed $[x]^\beta$ with its restriction to clause (c) [7]. He defined beta-reduction as follows:

The weak rules are the same along with one new rule (ξ_β), which states that

(ξ_β) $\qquad \dfrac{X \rhd Y}{[x]^\beta.X \rhd [x]^\beta.Y}$

Thus axiom schemes would be;

(**I**) $\mathbf{I}X \rhd X$;

(**K**) $\mathbf{K}XY \rhd X$;

(**S**) $\mathbf{S}XYZ \rhd XZ(YZ)$;

(ρ) $X \rhd X$.

The rules of inference are:

(μ) $\frac{X \rhd X`}{ZX \rhd ZX`}$

(ν) $\frac{X \rhd X`}{XZ \rhd X`Z}$

(τ) $\frac{X \rhd Y \quad Y \rhd Z}{X \rhd Z}$

(ξ_β) $\frac{X \rhd Y}{[x]^\beta .X \rhd [x]^\beta .Y}$

But his proposals had some problems:

(1) $X \rhd Y$ does not mean $X_\lambda \rhd_{\beta\eta} Y_\lambda$.

(2) Being functional is not preserved by reduction. That is, if we reduce a term which is functional, then it is not necessary that the term obtained after this reduction be functional. It can be a non-functional term also. Let us consider and example:

$$\mathbf{I}(xy) \rhd_w xy$$

$$[y]^\beta \mathbf{I}(xy) \rhd_\beta [y]^\beta (xy)$$

$$\mathbf{S}(\mathbf{KI})x \rhd_\beta \mathbf{S}(\mathbf{K}x)\mathbf{I}$$

$$\mathbf{S}(\mathbf{KI})x \rhd_\beta ([x]^\beta \mathbf{S}(\mathbf{K}x)\mathbf{I})x$$

$$\mathbf{S}(\mathbf{KI})x \rhd_\beta \mathbf{S}(\mathbf{S}(\mathbf{KS})\mathbf{K})(\mathbf{KI})x$$

Now, $\mathbf{S}(\mathbf{KI})x$ is functional but $\mathbf{S}(\mathbf{S}(\mathbf{KS})\mathbf{K})(\mathbf{KI})x$ is not.

(3) This proposal does not have a complete characterization of terms in normal form.

Example of a reduction under this proposal:

$$\mathbf{S}(\mathbf{KI})xy \rhd_w xy$$

$$[y]^\beta (xy) \equiv \mathbf{S}(\mathbf{K}x)\mathbf{I}$$

$$[x]^\beta \mathbf{S}(\mathbf{K}x)\mathbf{I} \equiv \mathbf{S}([x]^\eta \mathbf{S}(\mathbf{K}x))(\mathbf{KI})$$

by (ξ') twice $\equiv \mathbf{S}(\mathbf{S}(\mathbf{KS})\mathbf{K})(\mathbf{KI})$

So, $\mathbf{S}(\mathbf{KI}) \rhd_\beta \mathbf{S}(\mathbf{S}(\mathbf{KS})\mathbf{K})(\mathbf{KI})$

Under strong reduction $\mathbf{S}(\mathbf{KI}) \succ \mathbf{I}$, but it doesn't under this proposal.

Dr. Seldin's Proposal (unpublished)

Dr. J.P. Seldin also gave a proposal which uses $[x]^w$ (weak abstraction).

The definition of new reduction uses the weak rules, and

(ξ`) for all functional U except **S**, **K**, **I**, $Ux \rhd Y \Longrightarrow U \rhd [x]^w Y \quad (x \notin \mathrm{FV}(U))$;

(β_1) $\mathbf{S}(\mathbf{S}(\mathbf{K}X)\mathbf{I})Y \rhd \mathbf{S}XY$;

(β_2) **S***X*(**S**(**K***Y*)**I**) ⊳ **S***XY*;

(β_3) **S**(**K***U*)**I** ⊳ *U* for *U* functional.

Thus axiom schemes would be;

(**I**) **I***X* ⊳ *X*;

(**K**) **K***XY* ⊳ *X*;

(**S**) **S***XYZ* ⊳ *XZ*(*YZ*);

(ρ) *X* ⊳ *X*;

(β_1) **S**(**S**(**K***X*)**I**)*Y* ⊳ **S***XY*;

(β_2) **S***X*(**S**(**K***Y*)**I**) ⊳ **S***XY*;

(β_3) **S**(**K***U*)**I** ⊳ *U* for *U* functional.

The rules of inference are:

(μ) $\frac{X \rhd X`}{ZX \rhd ZX`}$

(ν) $\frac{X \rhd X`}{XZ \rhd X`Z}$

(τ) $\frac{X \rhd Y \quad Y \rhd Z}{X \rhd Z}$

(ξ`) for all functional *U* except **S**, **K**, **I**, $Ux \rhd Y \Longrightarrow U \rhd [x]^w Y \quad (x \notin \mathrm{FV}(U))$;

One problem with this proposal is that it also does not have a complete characterization of terms in normal form. Let us consider the following example:

$$\mathbf{S}(\mathbf{KI})xy \rhd_w \mathbf{KI}y(xy)$$

$$\rhd_w \mathbf{I}(xy)$$

$$\rhd_w xy$$

By (ξ') $\mathbf{S}(\mathbf{KI})x \rhd_\beta [y]^w(xy)$

$$\equiv \mathbf{S}([y]x)([y]y)$$

$$\equiv \mathbf{S}(\mathbf{K}x)\mathbf{I}$$

Hence by ξ again, $\mathbf{S}(\mathbf{KI}) \rhd_\beta [x]\mathbf{S}(\mathbf{K}x)\mathbf{I}$

$$\equiv \mathbf{S}([x]\mathbf{S}(\mathbf{K}x))([x]\mathbf{I})$$

$$\equiv \mathbf{S}(\mathbf{S}(\mathbf{KS})([x]\mathbf{K}x))(\mathbf{KI})$$

$$\equiv \mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{I})(\mathbf{KI})$$

$$\rhd_\beta \mathbf{S}(\mathbf{S}(\mathbf{KS})\mathbf{K})(\mathbf{KI})$$

Mezghiche's Proposal

The abstraction algorithm $[x]^{c\beta}X$ is defined by [18, Page 2],

$$[x]^{c\beta}X \equiv \mathbf{S}(\mathbf{K}([x]^{\eta}X))\mathbf{I}$$

The $\rhd_{c\beta}$ is the cβ-reduction which is the extension of weak combinatory reduction [18]. Mezghiche defines his new cβ-reduction by adding to the axioms of weak combinatory reduction the following two axioms:

(+) $\mathbf{S}(\mathbf{K}U)\mathbf{I} \rhd_{c\beta} U$ if U is functional

(ξ`) $Ux \rhd_{c\beta} Y \Longrightarrow U \rhd_{c\beta} [x]^{c\beta} Y$ if U is functional and $x \notin U$.

Thus axiom schemes would be;

(**I**) $\mathbf{I}X \rhd X$;

(**K**) $\mathbf{K}XY \rhd X$;

(**S**) $\mathbf{S}XYZ \rhd XZ(YZ)$;

(ρ) $X \rhd X$;

(+) $\mathbf{S}(\mathbf{K}U)\mathbf{I} \rhd_{c\beta} U$ if U is functional.

The rules of inference are:

(μ) $\frac{X \rhd X`}{ZX \rhd ZX`}$

(ν) $\frac{X \rhd X`}{XZ \rhd X`Z}$

(τ) $\frac{X \rhd Y \quad Y \rhd Z}{X \rhd Z}$

(ξ`) $Ux \rhd_{c\beta} Y \Longrightarrow U \rhd_{c\beta} [x]^{c\beta} Y$ if U is functional and $x \notin U$

The problem with this proposal is that it has only a partial characterization of terms in normal form.

Let us consider the following example:

$$\mathbf{S}(\mathbf{KI})xy \rhd_w \mathbf{KI}y(xy) \rhd_w \mathbf{I}(xy) \rhd_w xy$$

$$\mathbf{S}(\mathbf{KI}) \rhd_w [x]^{c\beta}\,[y]^{c\beta}\,(xy)$$

$$\equiv [x]^{c\beta}\,\mathbf{S}(\mathbf{K}([y]^{\eta}\,(xy))\mathbf{I}$$

$$\equiv [x]^{c\beta}\,\mathbf{S}(\mathbf{K}x)\mathbf{I}$$

$$\equiv \mathbf{S}(\mathbf{K}([x]^{\eta}\,\mathbf{S}(\mathbf{K}x)\mathbf{I}))\mathbf{I}$$

$$\equiv \mathbf{S}(\mathbf{K}(\mathbf{S}([x]^{\eta}\,(\mathbf{S}(\mathbf{K}x))(\mathbf{KI}))))\mathbf{I}$$

$$\equiv \mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{S}(\mathbf{KS})\mathbf{K})(\mathbf{KI})))\mathbf{I}$$

Because of the problem discussed above, it would be desirable to have a computer program in which a user has the power to define the definition/rules of abstraction/reduction, and the program would then generate examples. It takes a lot of time to generate examples by hand. If a computer program allowed the user to specify the abstraction and reduction by appropriate axiom schemes and rules, the program could then generate examples based on those rules. This would be useful to researchers who could then work on those examples and try to complete a solution to the problem of finding a reduction for CL equivalent to $\lambda\beta$-reduction. The next chapter discusses the program produced and the problems faced while trying to develop that program.

Chapter 4

SML/NJ and the Program

ML is a language that has some extremely interesting features. Its designers incorporated many modern programming-language ideas, yet the language is surprisingly easy to learn and use. ML is primarily a functional language, meaning that the basic mode of computation is the definition and application of functions. Functions can be defined by the user as in conventional languages, by writing code for the functions. But it is also possible in ML to treat functions as values and compute new functions from them with operators like function compositions. ML is a strongly typed language, meaning that all the values and variables have a type that can be determined at compile time (i.e., by examining the program but not running it). A value of one type cannot be given to a variable of another type. For example, the integer value 4 cannot be the value of a real-valued variable.

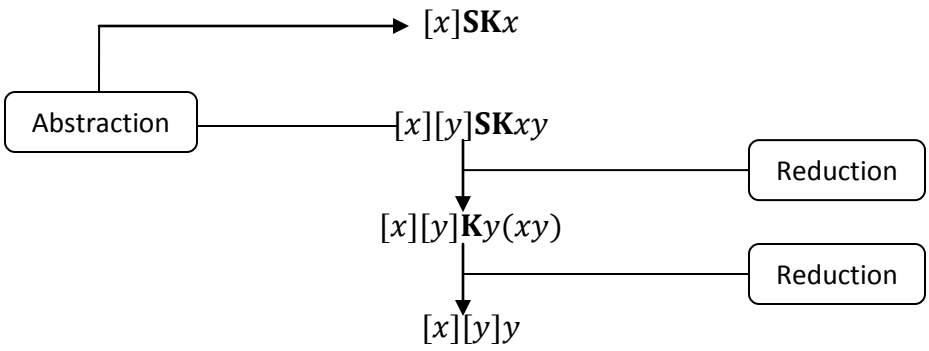
CL terms are defined through an inductive definition and functions are defined recursively. CL uses inductive and recursive definitions extensively (this is discussed in appendix A). This is the reason I chose ML for the program. I used SML/NJ (Standard ML/New Jersey) for MAC and SML/NJ for Windows for this program.

New method for expressing strong reduction

The way strong reduction was defined made it difficult to write an implementation program. While performing strong reduction, one does not only perform the reduction operation but also abstractions, so as one progresses, a number of abstractions and reductions are performed. The way these equations were written earlier didn't make it very clear as to when exactly a reduction operation was performed. Let us take the example that we had in chapter 3 for strong reduction, **SK** $\succ$ **KI**. Let us reduce this using both the old and the new method,

Old method:

SK



[x]**I**

KI

New method:

SK

$[x][[\mathbf{SK}]]_x$

$[x](\mathbf{SK}x)$

$[x][y][[\mathbf{SK}x]]_y$

$[x][y][[\mathbf{K}]]_y[[x]]_y$

$[x][y](\mathbf{K}y)(xy)$

Reduction

$[x][y]y$

$[x]\mathbf{I}$

KI

In first example, there are two reduction operations, **SK***xy* reduces to **K***y*(*xy*) and **K***y*(*xy*) reduces to *y*. But of these, only the second one is relevant. The reason the first is not relevant is, that when we do the expansion for *y* and get [*x*] [*y*] **SK***xy*, we have two choices. We can either perform the reduction operation and get **K***y*(*xy*). Since $\mathbf{S}xyz \rhd xz(yz)$, hence $\mathbf{SK}xy \rhd \mathbf{K}y(xy)$; where $x = \mathbf{K}$, $y = x$ and $z = y$. Alternatively we could perform the abstraction operation and get **SK***x* again. By the application of clause (c) as discussed in Chapter 2, Definition 2.7, $[x].Ux \equiv U$. Hence, $[x, y].(\mathbf{SK}x)y \equiv \mathbf{SK}x$; where $x = y$ and $U = \mathbf{SK}x$. While this is not a problem when doing it on paper, when one attempts to make a computer program, this would cause the program to go into an infinite

loop. Curry's linearization of strong reduction could be helpful here [5, Page 225]. Here, a type IIb step, which says $\mathbf{S}UV \Longrightarrow \lambda x.Ux(Vx)$, could replace the first contraction to obtain $[x][y]\mathbf{K}y(xy)$. This would identify the second contraction as the crucial one and would prevent the evaluation before it. This idea was captured by Dr. Robin Cockett who developed a semantic translation based on it. The semantic translation developed by him is as follows:

Axiom schemes:

$M \Rightarrow [x][[M]]_x$,

$[[\mathbf{S}MN]]_x \Rightarrow [[M]]_x[[N]]_x$,

$[[\mathbf{S}N]]_x \Longrightarrow \mathbf{S}Nx$,

$[[\mathbf{K}N]]_x \Rightarrow N$,

$[[\mathbf{S}]]_x \Rightarrow \mathbf{S}x$,

$[[\mathbf{K}]]_x \Rightarrow \mathbf{K}x$,

$[[U]]_x \Rightarrow Ux$.

Contraction steps:

$\mathbf{S}xyz \rhd xz(yz)$,

$\mathbf{K}xy \rhd x$,

$\mathbf{I}x \rhd x$,

$\mathbf{S}(\mathbf{K}x)\mathbf{I} \rhd x$,

$\mathbf{S}(\mathbf{K}x)(\mathbf{K}y) \rhd \mathbf{K}(xy)$.

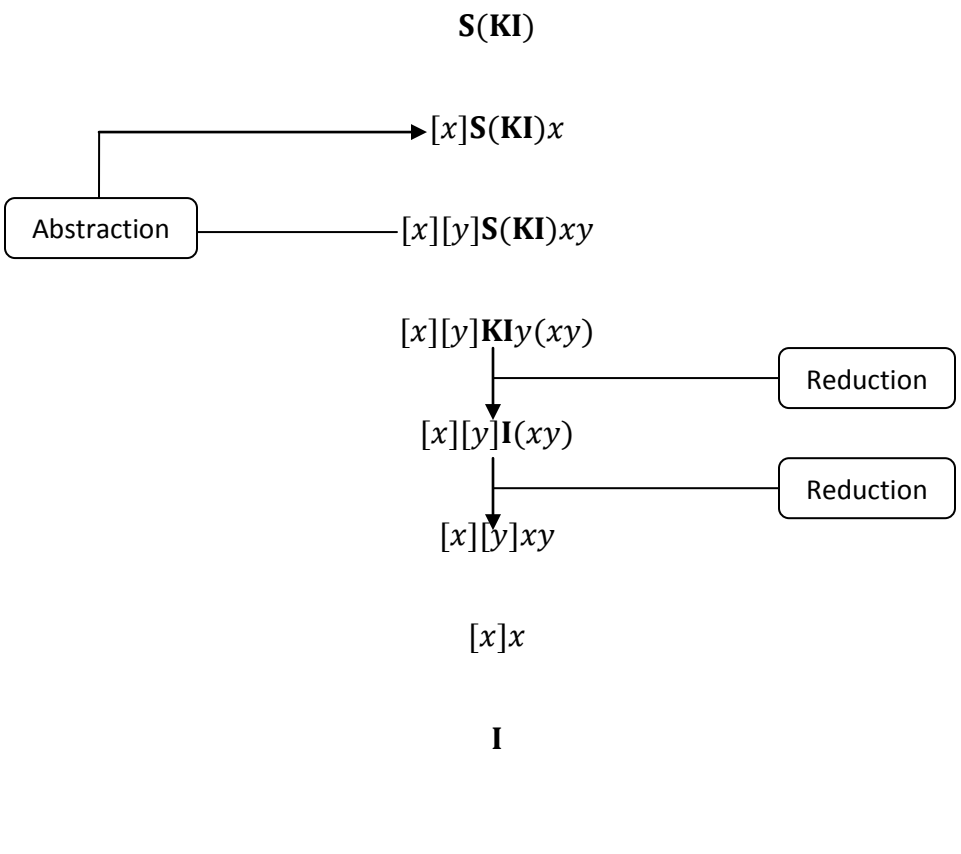
Hence, in the second method, when we get $[x][y][[\mathbf{SK}x]]_y$, we can tell the program to only perform a reduction operation and thus avoid infinitely looping. Also, the abstraction is not evaluated until either one of the contraction steps has occurred or no combinators are left to perform further contractions.

This new method is yet to be published.

Let us take a look at another example.

$\mathbf{S(KI)} \succ \mathbf{I}$

Old method:



New method:

$$\mathbf{S}(\mathbf{KI})$$

$$[x][[\mathbf{S}(\mathbf{KI})]]_x$$

$$[x]\mathbf{S}(\mathbf{KI})x$$

$$[x][y][[\mathbf{S}(\mathbf{KI})x]]_y$$

$$[x][y][[\mathbf{KI}]]_y[[x]]_y$$

$$[x][y]\mathbf{I}(xy)$$

Reduction

$$[x][y]xy$$

$$[x]x$$

$$\mathbf{I}$$

As we can see in the above example, once we get to $[x][y]\mathbf{S}(\mathbf{KI})xy$, we have two options; either to perform the reduction operation and obtain $[x][y]\mathbf{KI}y(xy)$, ($[x,y]\mathbf{S}(\mathbf{KI})xy \rhd [x,y]\mathbf{KI}y(xy)$ since $\mathbf{S}xyz \rhd xz(yz)$) or to perform the abstraction operation and obtain $[x][y]\mathbf{S}(\mathbf{KI})x$ back again (by the application of clause (c) which states that $[x].Ux \equiv U$) and enter an infinite loop. But in the new method, the computer program would clearly know that it needs to perform a reduction operation at this stage. Yet again, when we have $[x][y]\mathbf{I}(xy)$, we can either perform a reduction operation and get $[x][y]xy$ or perform an abstraction operation and obtain $[x][y]\mathbf{S}(\mathbf{KI})$ again and enter

an infinite loop. But with the new translation method, the only operation that we can perform at this point is reduction.

The program

Before I start the description of the program and the difficulties that I faced in generating it, I thank Dr. Robin Cockett, his post-doctorate student Brian Redmond, and his graduate student Sean Nichols. Without their help, this program could not have been written.

I start the program with defining datatypes [29] where variables, constants, **I**, **K**, **S** and application of a combinatory term to another combinatory term are defined. As discussed in Chapter 2, **I**, **K** and **S** are the combinators. The application of a combinatory term to another combinatory term is a recursive step. Dr. Robin Cockett then helped me in coding combinatory algebra reducer which defines all the basic reductions. There is also a pretty printer which uses the App, **I**, **K** and **S** defined above to print the result of reduction. User can type in examples and test the output by calling them. The `Lprint2` function prints the result of the strong reduction operation. Without this function, the result looks as follows;

```
- reducestrong (20, App (App (S, App (K, Var "x")), App (K, Var "y")));

val it = (0,App (K,App (#,#))) : int * CombTerm
```

but upon application of this function, the result is:

```
- Lprint2 (reducestrong (20, App (App (S, App (K, Var "x")), App (K, Var "y"))));

val it = "0: K(xy)" : string
```

The next step was to define some basic set operations: removing an element from a set, testing membership in a set, forming the union of two sets, and forming the intersection of two sets. These operations were then used in defining the set of free variables. Some of the set operations are not used but they are there should a user need to write code where he needs these set operations.

Next, a function named `Largest` is defined. This function finds the largest element in a given list. Then, a new function named `freshvars` was defined. Here, we use the `getOpt` function [28]. The way `getOpt` works is that it takes an option `` `a `` and returns some of `` `a ``. If there is nothing to return, it returns none.

Examples:

1. option int

   NONE

   SOME 5

   SOME 17

2. option string

   NONE

   SOME “xyz”

```
SOME “abc”
```

In order to perform strong reduction, we need to add variables at the end on the expression. But it is quite possible that the expression that we have already contains a variable. So we first have to scan the entire expression and find all the free variables in it.

In the program, we have used a naming scheme as “_1”, “_2” … etc. So we discard those variables that don’t fit the pattern “_n”. Now of the remaining variables, we remove the “_” so that “_n” will be “n” and then turn it into a number. In order to do this, we use `Int.fromString`. This changes the string to option int. We then use `getOpt`, with default = 0, to get int from this. Now, we find the largest number, add 1 to it, turn it back into a string and prepend “_”.

Let us consider an example to understand what exactly is happening here. Suppose we have an expression **SK***x*. So if we want to perform a strong reduction on this, we would add *y* to the end and a [*y*] to the front of it and then perform the operation. What this program would do is it would scan the given expression and find all the free variables. *x* is free in **SK***x*. So the next thing that it would do is discard those that don’t fit the pattern “_n”, in this case **S** and **K**. So now, we are left with one element which is *x* or “_1”. So it would then remove the “_” and take 1. It will then add 1 to it to get 2, turn it back to string and prepend “_” to it to get “_2”. This means that it would add *y* to the end.

Next, free variables and some translations from combinatory logic to lambda calculus and vice versa were defined. In order to do this, new datatypes and free variables had to be defined again. The next major thing to define was λ and CL abstraction and λ

and CL application to a term. Here, all the abstraction rules discussed in chapter 2 were defined.

Next, we proceeded with defining weak abstraction. Here, clause (c) from the definition (code) of `trans2b_abst` had to be removed. As discussed in chapter 3, weak abstraction is similar to abstraction in CL but with clause (c) omitted. Then, the definition of a functional term was coded in. A CL-term with one of the six forms **S***XY* (for some *X*, *Y*), **S***X*, **K***X*, **S**, **K**, **I**, is called functional or `fnl` [12, Definition 9.6]. Also, beta abstraction was defined which is very similar to CL abstraction defined earlier. We then created another function `trans2beta_abst_nf` in which we exclude the functional check. This is because one might need to attach terms which at certain times might not be functional.

There is a counter attached to the front of every reduction which tells the program how many times it should perform the reduction operation. When the counter reaches 0, the program stops the reduction and outputs the result.

Now a user can type in the expression and call the `reducestrong` function to perform the strong reduction. Here are a few examples with their outputs:

If the expression is **K***x* $\succ$ **K***x*, then it can be executed in one of the following three ways:

1. `val Strg_ex_1 = App (K,Var "x");`

Output;

```
val Strg_ex_1 = App (K,Var "x") : CombTerm
```

2. `reducestrong (20, App(K,Var "x"));`

Output;

```
val it = (0,App (K,Var "x")) : int * CombTerm
```

3. `Lprint2 (reducestrong (20, App (K, Var "x")));`

Output;

```
val it = "0: Kx" : string
```

Similarly, for **SK** $\succ$ **KI**;

1. `val Strg_ex_2 = App (S, K);`

Output;

```
val Strg_ex_2 = App (S,K) : CombTerm
```

2. `reducestrong (15, App (S, K));`

Output;

```
val it = (0,App (K,I)) : int * CombTerm
```

3. `Lprint2 (reducestrong (15, App (S, K)));`

Output;

```
val it = "0: KI" : string
```

For **S**(**K***x*)(**K***y*) $\succ$ **K**(*xy*);

1. `val Strg_ex_3 = App (App (S, App (K, Var "x")), App (K, Var "y"));`

Output;

```
val Strg_ex_3 = App (App (S,App #),App (K,Var #)) : CombTerm
```

2. `reducestrong (25, App (App (S, App (K, Var "x")), App (K, Var "y")));`

Output;

```
val it = (0,App (K,App (#,#))) : int * CombTerm
```

3. `Lprint2 (reducestrong (25, App (App (S, App (K, Var "x")), App (K, Var "y"))));`

Output;

```
val it = "0: K(xy)" : string
```

For **S(KI)** $\succ$ **I**;

1. `val Strg_ex_4 = App (S, App (K, I));`

Output;

```
val Strg_ex_4 = App (S,App (K,I)) : CombTerm
```

2. `reducestrong (20, App (S, App (K, I)));`

Output;

```
val it = (0,I) : int * CombTerm
```

3. `Lprint2 (reducestrong (20, App (S, App (K, I))));`

Output;

```
val it = "0: I" : string
```

For **S**(**KS**)(**S**(**KK**)) ≻ **K**;

1. `val Strg_ex_5 = App (App (S, App (K, S)), App (S, App (K, K)));`

Output;

```
val Strg_ex_5 = App (App (S,App #),App (S,App #)) : CombTerm
```

2. `reducestrong (20, App (App (S, App (K, S)), App (S, App (K, K))));`

Output;

```
val it = (0,K) : int * CombTerm
```

3. `Lprint2 (reducestrong (20, App (App (S, App (K, S)), App (S, App (K, K)))));`

Output;

```
val it = "0: K" : string
```

For **S**(**S**(**KS**)(**S**(**KK**)**K**))(**K**(**SKK**)) ≻ **K**;

1. `val Strg_ex_6 = App (App (S, App (App (S, App (K, S)), App (App (S, App(K, K)), K))), App (K, App(App (S, K), K)));`

Output;

```
val Strg_ex_6 = App (App (S,App #),App (K,App #)) : CombTerm
```

2. `reducestrong (20, App (App (S, App (App (S, App (K, S)), App (App (S, App(K, K)), K))), App (K, App(App (S, K), K))));`

Output;

```
val it = (0,K) : int * CombTerm
```

3. `Lprint2 (reducestrong (20, App (App (S, App (App (S, App (K, S)), App (App (S, App(K, K)), K))), App (K, App(App (S, K), K)))));`

Output;

```
val it = "0: K" : string
```

References

[1] Barendregt H., *The Lambda Calculus – Its syntax and Semantics*, Elsevier Science Publishers B.V., 1984.

[2] Bunder M., Hindley R. and Seldin J., On Adding (ξ) to Weak Equality in Combinatory Logic, *The Journal of Symbolic Logic*, Volume 54, Number 2, Pages 590-607, 1989.

[3] Cagman N, Hindley R., Combinatory Weak Reduction in Lambda Calculus, *Theoretical Computer Science*, Volume 198, Pages 239-247, 1998.

[4] Church A., *The Calculi of Lambda-Conversion*, Princeton University Press, 1941.

[5] Curry H., Feys R. and Craig W, *Combinatory Logic Volume 1*, North-Holland Publishing Company-Amsterdam, 1968.

[6] Curry H., Hindley R. and Seldin J., *Combinatory Logic Volume 2*, North-Holland Publishing Company-Amsterdam, 1972.

[7] Curry H., Hindley R. and Seldin J., Beta Strong Reduction in Combinatory Logic, *The Journal of Symbolic Logic*, Volume 49, Number 2, Pages 688-689, 1984.

[8] Eisenbach S., *Functional Programming Languages, Tools and Architectures*, Ellis Horwood Limited, 1987.

[9] Hindley R., Axioms for Strong Reduction in Combinatory Logic, *The Journal of Symbolic Logic*, Volume 32, Number 2, Pages 224-236, 1967.

[10] Hindley R., Combinatory Reductions and Lambda Reductions Compared, *Zeitschr. j. math. Logik und Grundlagen d. Math.*, Volume 23, Pages 169-180, 1977.

[11] Hindley R., *The Church-Rosser Property and a Result in Combinatory Logic*, PhD. Dissertation, University of Newcastle, 1964.

[12] Hindley R. and Seldin J., *Introduction to Combinators and Lambda-Calculus*, Cambridge University Press, 2008.

[13] Klop J., *Combinatory Reduction Systems*, Dissertation, University of Amsterdam, Mathematisch Centrum, 1980.

[14] Lercher B., Lambda-calculus Terms That Reduce to Themselves, *Notre Dame Journal of Formal Logic*, Volume XVII, Number 2, Pages 291-292, 1976.

[15] Lercher B., Strong Reduction and Normal Forms in Combinatory Logic, *The Journal of Symbolic Logic*, Volume 32, Number 2, Pages 213-223, 1967.

[16] Lercher B., *Strong Reduction and Recursion in Combinatory Logic*, PhD. Dissertation, The Pennsylvania State University, 1963.

[17] Lercher B., The Decidability of Hindley's Axioms for Strong Reduction, *The Journal of Symbolic Logic*, Volume 32, Number 2, Pages 237-239, 1967.

[18] Mezghiche M., Note on Pseudo-cβ Normal Form in Combinatory Logic, *Theoretical Computer Science*, Volume 66, Pages 323-331, 1989.

[19] Mezghiche M., cβ-Machine with λβ-reduction, *Theoretical Computer Science*, Volume, 189, Pages 221-228, 1997.

[20] Paulson L., *ML for the Working Programmer*, Cambridge University Press, 1991.

[21] Rosser B., Highlights of the History of the Lambda-Calculus, *Annals of the History of Computing*, Volume 6, Number 4, Pages 337-349, 1984.

[22] Schönfinkel M., On the Building Blocks of Mathematical Logic, van Heijenoort, *From Frege to Gödel A Source Book in Mathematical Logic*, 1879-1931, Harvard University Press, 1977.

[23] Seldin J., On List and Other Abstract Data Types in the Calculus of Construction, *Math. Struct. In Comp. Science*, vol. 10, *Cambridge University Press*, Pages 261-276, 2000.

[24] Ullman J., *Elements of ML Programming, ML 97 Edition*, Prentice Hall, 1997.

[25] Ullman J., *Fundamental Concepts of Programming Systems*, Addison-Wesley Publishing Company, 1976.

[26] Baudinet M. and MacQueen D., Tree Pattern Matching for ML (extended abstract), http://www.smlnj.org/compiler-notes/85-note-baudinet.ps, as accessed on 01-04-2009.

[27] Barendregt H. and Barendsen E., *Introduction to Lambda Calculus*, http://www.cs.ru.nl/E.Barendsen/onderwijs/sl2/materiaal/lambda.pdf, as accessed on 02-04-2009.

[28] http://www.smlnj.org/index.html as accessed on 02-06-2008.

[29] http://www.standardml.org/Basis/date.html, as accessed on 04-04-2009.

Appendix A

Short tutorial for SML/NJ

Standard ML is a safe, modular, strict, functional, polymorphic programming language with compile-time type checking and type inference, garbage collection, exception handling, immutable data types and updatable references, abstract data types, and parametric modules. It has efficient implementations and a formal definition with a proof of soundness [28].

To run SML/NJ in interactive mode [24, Chapter 1], in response to the command prompt type

```
sml
```

SML/NJ will respond with:

```
Standard ML of New Jersey…
-
```

The dash in the second line is ML's prompt. The prompt invites us to type an expression, and ML will respond with the value of the expression.

When we are in interactive mode, the simplest thing we can do is type an expression in response to the ML prompt (-). ML will respond with value and its type.

Example: Here is an example of an expression that we may type and the ML response.

```
1+2*3;
```

val it = 7 : int

Here, we have typed the expression 1 + 2 * 3, and ML responds that the value of variable it is 7, and that the type of this value is integer. The variable '`it`' plays a special role in ML. It receives the value of any expression that we type. Two useful points to observe are;

- An expression must be followed by a semicolon to tell the ML system that the instruction is finished. If ML expects more input when a <return> is typed, it will respond with the prompt = instead of -. The = sign is a warning that we have not finished our input expression.
- The response of ML to an expression is:
    1. The word val standing for "value,"
    2. The variable name it, which stands for the previous expression,
    3. An equal sign,
    4. The value of expression (7 in this example),
    5. A colon, which in ML is the symbol that associates a value with its type, and
    6. An expression that denotes the type of the value. In our example, the value of the expression is an integer, so the type int follows the colon.

The keyword fun introduces function definitions. The simplest form of function declaration is

fun<identifier>(<parameter list>) = <expression>;

That is, the keyword fun is followed by the name of the function, a list of the parameters for that function, an equal-sign, and an expression involving the parameters. This expression becomes the value of the function when we give the function arguments to correspond to its parameters.

Example:

```
fun square(x:real) = x*x;
```

val square = fn : real → real

The function square has one parameter, x. By following parameter x with a colon and the type real, we declare to ML that the parameter of function square is of type real. ML then infers that the expression x*x represents real multiplication, and therefore the value returned by square is of type real.

It is necessary to indicate that x is real somewhere. Otherwise, ML will use a colon and type integer, for x, resulting in a function that can square integers but not reals:

```
fun square(x) = x*x;
```

val square = fn : int → int

As an example of the use of the square function, suppose we have defined the variable pi and radius to have values 3.14159 and 4.0, as in previous example.

```
pi*square(radius);
```

val it = 50.26544

Following is the program that produces the largest of three real numbers.

```
fun max3(a:real, b, c) = (* maximum of three reals *)
     if a>b then
        if a>c then a
        else c
      else
        if b>c then b
        else c;
```

*val max3 = fn : real * real * real → real*

val t = (1.0, 2.0, 3.0);

max3(t);

ML produces the value 3.0.

Here is an example which uses the let function which is used in the program:

```
fun factorial n =
 let
     fun tail_fact p 0 = p
       | tail_fact p m = tail_fact (p * m) (m - 1)
 in
     tail_fact 1 n
 end
```

All these programs are of type-and-execute fashion i.e. we type the program on the ML window and ML executes them there itself. But there is a way by which we can load and execute ML programs previously saved onto our hard drive. To do this, we use the 'use' command. Its syntax is –use"<program name>.sml"; [28]. We can type ML programs in Notepad (Windows) or TextEdit (Macintosh), but while saving, we have to save them with .sml extension.

Recursive Functions

It is possible for ML functions to be recursive (as mentioned at the start of this chapter), that is defined in terms of themselves, either directly or indirectly [24, Chapter 1]. Normally a recursive function consists of

1. A *basis*, where for sufficiently small arguments we compute the result without making any recursive calls, and
2. An *inductive step*, where for arguments not handled by the basis, we call the function recursively, one or more times, with smaller arguments.

Example

Let us write a function reverse(L) that produces the reverse of the list L^3. For example, reverse([1,2,3]) produces the list [3,2,1].

BASIS: The basis is the empty list; the reverse of the empty list is the empty list.

INDUCTION: For the inductive step, suppose *L* has at least one element. Let the first or head element of *L* be *h*, and let the tail or remaining elements of *L* be the list *T*. Then we can construct the reverse of list *L* by reversing *T* and following it by the element *h*.

For instance, if *L* is [1,2,3], then *h* = 1, *T* is [2,3], the reverse of *T* is [3,2], and the reverse of *T* concatenated with the list containing only *h* is [3,2]@[1], or [3,2,1].

```
fun reverse(L) =
        if L = nil then nil
        else reverse(tl(L)) @ [hd(L)];
```

val reverse = fn : 'a list → 'a list

We see the ML definition of reverse that follows the basis and inductive step.

Now, let us revisit the definition of term that we presented in Chapter 2. The set of expressions called *CL-terms* is defined inductively as follows:

(c) All variables and atomic constants, including **I**, **K**, **S**, are CL-terms.

(d) If *X* and *Y* are CL-terms, then so is (*XY*).

Since term is a recursive definition, it should easily be coded in ML, i.e. it should be fairly easy to write an ML program of it.

Exiting the interactive system

Typing control-D (EOF) at top level will cause an exit to the shell (or the parent process from which sml was run). One can also terminate by calling OS.Process.exit(OS.Process.success).

Appendix B

The program (code)

```
datatype CombTerm = Var of string
                  | Const of string
                  | I | S | K
                  | App of CombTerm * CombTerm;


fun isVar (Var _) = true
  | isVar _ = false;


fun isConst (Const _) = true
  | isConst _ = false;



(* Here is a little combinatory algebra reducer *)
fun reduce (n,t)
   = case (n,t) of
        (0,t) => (0,t)
      | (n,App (App(K,t1),_)) => reduce (n-1,t1)
      | (n,App (App (App (S,t1),t2),t3)) =>
                        reduce (n-
1,App(App(t1,t3),App(t2,t3)))
      | (n,App (I,t1)) => reduce (n-1,t1)
      | (n,App(t1,t2)) => (case (reduce (n,t1)) of
```

```
                          (m,t1') => (case (reduce (m,t2))
of
                              (m',t2') =>
                                  if n > m' then reduce
(m',App(t1',t2'))
                                  else (m',App(t1',t2'))))
      | (n,S) => (n,S)
      | (n,K) => (n,K)
      | (n,I) => (n,I)
      | (n,Var s)   => (n,Var s)
      | (n,Const s) => (n,Const s);


(* A pretty printer *)
fun Lprint I = "I"
  | Lprint S = "S"
  | Lprint K = "K"
  | Lprint (Const s) = s
  | Lprint (Var s) = s
  | Lprint (App (t1, App(t2,t3))) = (Lprint t1)^"("^(Lprint
(App(t2,t3)))^")"
  | Lprint (App (t1,t2)) = (Lprint t1)^(Lprint t2);


fun Lprint2 (n, t) = (Int.toString n) ^ ": " ^ (Lprint t);


(*   Examples *)
```

```
reduce (4,App(App(App(S,K),K),Var "x"));

reduce (4,App(Const "a",App(App(K,Var "x"),S)));


val Omega = App(App(App(S,I),I),App(App(S,I),I));


Lprint ((fn (x,y) => y) (reduce (3,Omega)));


val t1 = App(App(App(S,App(K,Var "x")),App(K,Var
"y")),Omega);


fun Preduce n t = Lprint ((fn (x,y) => y) (reduce(n, t)));




(*************************************************************
*******

 *

 *   Some basic utilities for handling sets:

 *

 * Removing an element from a set (represented as a list)

*************************************************************
******)


(* Removing an element from a set (represented as a list)
*)

fun remove v [] = []

  | remove v (y :: ys) =
```

```
                if y = v then remove v ys
                else y ::(remove v ys);

(* testing membership in a set *)
fun member x [] = false
  | member x (y::ys) = if x=y then true
                       else (member x ys);

(* Forming the union of two sets *)
fun union [] ys = ys
  | union xs [] = xs
  | union (x::xs) ys =
          if (member x ys) then (union xs ys)
          else x::(union xs ys);

(* Forming the intersection of two sets *)

fun intersection [] ys = []
  | intersection (x::xs) ys =
             if (member x ys) then x::(intersection xs ys)
             else (intersection xs ys);

(* Filter elements out of a list *)

fun filter f [] = []
```

```
  | filter f (x::xs) = if (f x) then x::(filter f xs)
                                 else filter f xs;


(*************************************************************
*****
 *
 *  To calculate the free variables of a term
 *

*************************************************************
****)


fun freevars (Var x) = [x]
  | freevars (App (t1,t2)) = union (freevars t1) (freevars
t2)
  | freevars _ = [];


fun largest [] = 0
  | largest (x::xs) = let val y = largest xs
    in
        if x > y then x else y
    end;


fun freshvar t = "_" ^ Int.toString (largest (map
                     (fn s =>
getOpt(Int.fromString(String.extract (s, 1, NONE)), 0))
                     (filter (fn s => String.extract (s, 0,
SOME 1) = "_") (freevars t))) + 1);
```

```
(*************************************************************
***********

*************************************************************
**********

 *                    TRANSLATIONS
*

*************************************************************
**********

*************************************************************
**********)


(* First we need a corresponding datatype for lambda terms
*)


datatype LambdaTerm = LVar of string
                    | LAbst of (string * LambdaTerm)
                    | LApp of (LambdaTerm * LambdaTerm)
                    | LConst of string;


fun freeLvars (LVar x) = [x]
  | freeLvars (LAbst(x,t)) = remove x (freeLvars t)
  | freeLvars (LApp(t1,t2)) = union (freeLvars t1)
(freeLvars t2)
  | freeLvars _ = [];
```

```
(* the easy translation first from combinatory logic to the
lambda

calculus: *)


fun trans_C2L (Var x) = LVar x

  | trans_C2L (Const c) = LConst c

  | trans_C2L I = LAbst ("x",LVar "x")

  | trans_C2L K = LAbst ("x",LAbst ("y",LVar "x"))

  | trans_C2L S = LAbst ("x",LAbst ("y",LAbst ("z"

              ,LApp (LApp (LVar "x",LVar "z"),LApp (LVar
"y",LVar "z")))))

  |  trans_C2L (App(t1,t2)) = LApp(trans_C2L t1,trans_C2L
t2);


(* Now a little more challenging: the translation from
lambda

 * calculus to combinatory logic -- recall there is more
than one

 * possible translation!!  Here is the simplest tarnslation
... *)


(* translating abstraction ... *)

fun trans_abst x (Var y) = if x=y then I

                           else App(K,Var y)

  | trans_abst x (App(c1,c2)) = App(App(S,trans_abst x
c1),trans_abst x c2)

  | trans_abst x v = App(K,v)
```

```
fun trans_L2C (LVar x) = (Var x)

  | trans_L2C (LConst c) = (Const c)

  | trans_L2C (LAbst(x,t)) = trans_abst x (trans_L2C t)

  | trans_L2C (LApp(t1,t2)) = App(trans_L2C t1,trans_L2C
t2);


(* Here is the other translation which tests to see whether
variables

 * are free .. *)


fun trans2b_abst x (Var y) = if x=y then I

                             else App(K,Var y)

  | trans2b_abst x (App(c1,c2)) = App(App(S,trans2_abst x
c1),trans2_abst x c2)

  | trans2b_abst x v = App(K,v)

and trans2a_abst x (App(u,Var y))

    = if x=y andalso not (member x (freevars u)) then u

      else trans2b_abst x (App(u,Var y))

  | trans2a_abst x n =  trans2b_abst x n

and trans2_abst x n = if not (member x (freevars n)) then
App(K,n)

                      else trans2a_abst x n;


fun trans2_L2C (LVar x) = (Var x)

  | trans2_L2C (LConst c) = (Const c)

  | trans2_L2C (LAbst(x,t)) =  trans2_abst x (trans2_L2C t)
```

```
  | trans2_L2C (LApp(t1,t2)) = App(trans2_L2C t1,trans2_L2C
t2);




(* examples *)


val Puff = LAbst("x",LApp(LVar "x",LVar "x"));

val LOmega = LApp(Puff,Puff);

val fPuff = LAbst("x",LApp(LApp(LVar "f",LVar "x"),LVar
"x"));

val Y = LAbst("f",LApp(fPuff,fPuff));


fun pL2C x = print((Lprint (trans_L2C x))^"\n");

fun p2L2C x = print((Lprint (trans2_L2C x))^"\n");




(****Weak Abstraction****)


fun trans2w_abst x (Var y) = if x=y then I

                             else App(K,Var y)

  | trans2w_abst x (App(c1,c2)) = App(App(S,trans2_abst x
c1),trans2_abst x c2)

  | trans2w_abst x v = App(K,v)

and trans2a_abst x (App(u,Var y))
```

```
    = trans2w_abst x (App(u,Var y))
  | trans2a_abst x n =  trans2b_abst x n
and trans2_abst x n = if not (member x (freevars n)) then
App(K,n)
                      else trans2w_abst x n;



(*******Functional Term*******)

fun fnl (App(App(S, x), y)) = true
            | fnl (App(S, x)) = true
            | fnl (App(K, x)) = true
            | fnl S = true
            | fnl K = true
            | fnl I = true
            | fnl x = false;



(******Beta Abstraction*****)

fun trans2beta_abst x (Var y) = if x=y then I
                                else App(K,Var y)
  | trans2beta_abst x (App(c1,c2)) = App(App(S,trans2_abst
x c1),trans2_abst
```

```
x c2)

  | trans2beta_abst x v = App(K,v)

and trans2a_abst x (App(u,Var y))

    = if x=y andalso not (member x (freevars u)) andalso
fnl u then u

      else trans2beta_abst x (App(u,Var y))

  | trans2a_abst x n =  trans2beta_abst x n

and trans2_abst x n = if not (member x (freevars n)) then
App(K,n)

                      else trans2a_abst x n;


fun trans2_L2C (LVar x) = (Var x)

  | trans2_L2C (LConst c) = (Const c)

  | trans2_L2C (LAbst(x,t)) =  trans2_abst x (trans2_L2C t)

  | trans2_L2C (LApp(t1,t2)) = App(trans2_L2C t1,trans2_L2C
t2);




(* Same as above, but without the "is functional" check in
Clause C ("_nf" stands for non functional) *)


fun trans2beta_abst_nf x (Var y) = if x=y then I

                                 else App(K,Var y)

  | trans2beta_abst_nf x (App(c1,c2)) =
App(App(S,trans2_abst_nf x c1),trans2_abst_nf x c2)

  | trans2beta_abst_nf x v = App(K,v)

and trans2a_abst_nf x (App(u,Var y))
```

```
    = if x=y andalso not (member x (freevars u)) then u
      else trans2beta_abst_nf x (App(u,Var y))
  | trans2a_abst_nf x n =  trans2beta_abst_nf x n
and trans2_abst_nf x n = if not (member x (freevars n))
then  App(K,n)
                     else trans2a_abst_nf x n;



(*******Strong Reduction********)


fun reducestrong (0,t) = (0,t)
  | reducestrong (n,App (App(K,t1),_)) = reducestrong (n-
1,t1)
  | reducestrong (n,App (App (App (S,t1),t2),t3)) =
                       reducestrong (n-
1,App(App(t1,t3),App(t2,t3)))
  | reducestrong (n,App (I,t1)) = reducestrong (n-1,t1)
  | reducestrong (n,App(t1,t2)) = (case (attempt_noxi
(n,t1)) of
                            (m,t1') => (case (attempt_noxi
(m,t2)) of
                                (m',t2') =>
                                    if n > m' then
reducestrong (m',App(t1',t2'))
                                              else xirule
(n, App(t1,t2))))                   (* (m',App(t1',t2'))))
*)
  | reducestrong (n,S) = (n,S)
```

```
  | reducestrong (n,K) = (n,K)

  | reducestrong (n,I) = (n,I)

  | reducestrong (n,Var s)   = (n,Var s)

  | reducestrong (n,Const s) = (n,Const s)

and attempt_noxi (0,t) = (0,t)

  | attempt_noxi (n,App (App(K,t1),_)) = attempt_noxi (n-
1,t1)

  | attempt_noxi (n,App (App (App (S,t1),t2),t3)) =

                       attempt_noxi (n-
1,App(App(t1,t3),App(t2,t3)))

  | attempt_noxi (n,App (I,t1)) = attempt_noxi (n-1,t1)

  | attempt_noxi (n,App(t1,t2)) = (case (attempt_noxi
(n,t1)) of

                           (m,t1') => (case (attempt_noxi
(m,t2)) of

                               (m',t2') =>

                                   if n > m' then
attempt_noxi (m',App(t1',t2'))

                                              else
(m',App(t1',t2'))))

  | attempt_noxi (n,S) = (n,S)

  | attempt_noxi (n,K) = (n,K)

  | attempt_noxi (n,I) = (n,I)

  | attempt_noxi (n,Var s)   = (n,Var s)

  | attempt_noxi (n,Const s) = (n,Const s)

and xirule (n, t)

   = let val fv = freshvar t

         val y = reducexi (n-1, App (t, Var fv))
```

```
     in
         case y of (k, t2) => if (k = ~1)
                              then (n,t)
(*                              else (k,t2) *)
                              else let val z = reducestrong (k,
t2)
                                   in
                                       case z of (m, t3) => (0,
trans2_abst_nf fv t3)
                                   end
     end
and reducexi (0,t) = (~1,t)
  | reducexi (n,App (App(K,t1),_)) = (n,t1)
  | reducexi (n,App (App (App (S,t1),t2),t3)) =
                              (n,App(App(t1,t3),App(t2,t3)))
  | reducexi (n,App (I,t1)) = (n,t1)
  | reducexi (n,App(t1,t2)) = xirule (n, App(t1,t2));


(*
 * Note in the last case above, the Xi (Xi prime actually)
rule is:
 *    if Ux >- Y then U >- [x].Y (for x not in FV(U))
 *  we get to pick our var. x, so we just pick it such
 *   that it is not free in U, therefore we have only
 *   to check the other condition (that Ux >- Y)
 *)
```

```
(* Some examples to use with Strong Reduction *)


val Strg_ex_1 = App (K,Var "x");

    (* Kx >- Kx *)

val Strg_ex_2 = App (S, K);

    (* SK >- KI *)

val Strg_ex_3 = App (App (S, App (K, Var "x")), App (K, Var
"y"));

    (* S(Kx)(Ky) >- K(xy) *)

val Strg_ex_4 = App (S, App (K, I));

    (* S(KI) >- I *)

val Strg_ex_5 = App (App (S, App (K, S)), App (S, App (K,
K)));

    (* S(KS)(S(KK)) >- K *)

val Strg_ex_6 = App (App (S, App (App (S, App (K, S)), App
(App (S, App(K, K)), K))), App (K, App(App (S, K), K)));

    (* S(S(KS)(S(KK)K))(K(SKK)) >- K *)
```

The Output

```
Standard ML of New Jersey, Version 110.0.7, September 28,
2000 [CM&CMB]

- use "D:\\prog-25mar.sml";

[opening D:\prog-25mar.sml]

datatype CombTerm

  = App of CombTerm * CombTerm | Const of string | I | K |
S | Var of string

val isVar = fn : CombTerm -> bool

val isConst = fn : CombTerm -> bool

GC #0.0.0.0.1.15:   (0 ms)

val reduce = fn : int * CombTerm -> int * CombTerm

val Lprint = fn : CombTerm -> string

val Lprint2 = fn : int * CombTerm -> string

val it = (2,Var "x") : int * CombTerm

val it = (3,App (Const "a",Var "x")) : int * CombTerm

val Omega = App (App (App #,I),App (App #,I)) : CombTerm

val it = "SII(SII)" : string

val t1 = App (App (App #,App #),App (App #,App #)) :
CombTerm

val Preduce = fn : int -> CombTerm -> string

GC #0.0.0.0.2.60:   (0 ms)

val remove = fn : ''a -> ''a list -> ''a list

val member = fn : ''a -> ''a list -> bool

val union = fn : ''a list -> ''a list -> ''a list

val intersection = fn : ''a list -> ''a list -> ''a list
```

```
val filter = fn : ('a -> bool) -> 'a list -> 'a list
val freevars = fn : CombTerm -> string list
val largest = fn : int list -> int
val freshvar = fn : CombTerm -> string
datatype LambdaTerm
  = LAbst of string * LambdaTerm
  | LApp of LambdaTerm * LambdaTerm
  | LConst of string
  | LVar of string
val freeLvars = fn : LambdaTerm -> string list
GC #0.0.0.0.3.129:   (0 ms)
val trans_C2L = fn : CombTerm -> LambdaTerm
val trans_abst = fn : string -> CombTerm -> CombTerm
val trans_L2C = fn : LambdaTerm -> CombTerm
val trans2b_abst = fn : string -> CombTerm -> CombTerm
val trans2a_abst = fn : string -> CombTerm -> CombTerm
val trans2_abst = fn : string -> CombTerm -> CombTerm
val trans2_L2C = fn : LambdaTerm -> CombTerm
val Puff = LAbst ("x",LApp (LVar #,LVar #)) : LambdaTerm
val LOmega = LApp (LAbst ("x",LApp #),LAbst ("x",LApp #)) :
LambdaTerm
val fPuff = LAbst ("x",LApp (LApp #,LVar #)) : LambdaTerm
val Y = LAbst ("f",LApp (LAbst #,LAbst #)) : LambdaTerm
val pL2C = fn : LambdaTerm -> unit
val p2L2C = fn : LambdaTerm -> unit
GC #0.0.0.0.4.197:   (0 ms)
```

```
val trans2w_abst = fn : string -> CombTerm -> CombTerm

val trans2a_abst = fn : string -> CombTerm -> CombTerm

val trans2_abst = fn : string -> CombTerm -> CombTerm

val fnl = fn : CombTerm -> bool

val trans2beta_abst = fn : string -> CombTerm -> CombTerm

val trans2a_abst = fn : string -> CombTerm -> CombTerm

val trans2_abst = fn : string -> CombTerm -> CombTerm

val trans2_L2C = fn : LambdaTerm -> CombTerm

val trans2beta_abst_nf = fn : string -> CombTerm ->
CombTerm

val trans2a_abst_nf = fn : string -> CombTerm -> CombTerm

val trans2_abst_nf = fn : string -> CombTerm -> CombTerm

GC #0.0.0.0.5.275:   (0 ms)

D:\prog-25mar.sml:260.1-307.53 Warning: match nonexhaustive

          (0,t) => ...

          (n,App (App (<pat>,<pat>),_)) => ...

          (n,App (App (<pat>,<pat>),t3)) => ...

          (n,App (I,t1)) => ...

          (n,App (t1,t2)) => ...


val reducestrong = fn : int * CombTerm -> int * CombTerm

val attempt_noxi = fn : int * CombTerm -> int * CombTerm

val xirule = fn : int * CombTerm -> int * CombTerm

val reducexi = fn : int * CombTerm -> int * CombTerm

val Strg_ex_1 = App (K,Var "x") : CombTerm

val Strg_ex_2 = App (S,K) : CombTerm
```

```
val Strg_ex_3 = App (App (S,App #),App (K,Var #)) :
CombTerm

val Strg_ex_4 = App (S,App (K,I)) : CombTerm

val Strg_ex_5 = App (App (S,App #),App (S,App #)) :
CombTerm

val Strg_ex_6 = App (App (S,App #),App (K,App #)) :
CombTerm

val it = () : unit

-
```