MAC TR-136


FUNCTIONAL DOMAINS OF APPLICATIVE LANGUAGES


Stephen A. Ward


September 1974

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT   MAC


CAMBRIDGE                                              MASSACHUSETTS 02139

# ACKNOWLEDGEMENT

# FUNCTIONAL DOMAINS OF APPLICATIVE LANGUAGES

## Abstract

The expressive power of a particular applicative language
may be characterized by the set of abstract functions di-
rectly representable in that language.  The common FUNARG
and applicative order problems are scrutinized in this
way, and the effects of these weaknesses are related to the
inexpressibility of classes of functions.

Certain computable functions which are inexpressible in the
lambda calculus are identified, and it is established that
the interpretation of these functions requires a mechanism
fundamentally equivalent to multiprocessing.  The EITHER
construct is proposed as an extension to the lambda calculus,
and several theories including this mechanism are presented
and proved consistent (in the sense that they introduce no
new equivalences into the lambda calculus).

A syntactic analog to the Scott construction, *-conversion,
is developed in conjunction with these theories; this adjunct
allows reduction of expressions having no normal forms in
the usual lambda calculus to finite normal form approximations
of the expressions.  This leads naturally to a technique for
proving the extensional equivalence of lambda calculus
expressions which are not interconvertible.

# Table of Contents

Table of Contents

*This empty page was substituted for a blank page in the original document.*

Chapter 1:

Introduction

## 1.1: Programming Language Semantics

The semantics of a programming language may be viewed as a theory which
accounts for the behavior of programs written in that language. An
interpreter for a language L is a model for the semantics of L, and a language
whose semantics is incomplete (in the sense of an incomplete theory) may have
many "correct" interpreters which behave differently just as an incomplete
theory may have disparate models. We find that the usual more specific
definitions of semantics (e.g. "the relation between expressions and the
objects which they denote") make assumptions about the structure of a universe
of "meanings" which are difficult to justify in the general case, where side
effects, assignment, and transfers of control must be accounted for
semantically. Such considerations motivate the restriction of the present
work to applicative languages.

Serious concern for formal semantics is not usually an important consideration
in the architecture of practical languages. Typically a language is designed
largely by pragmatic considerations and the formal statement of its semantics
is either abandoned entirely or postponed until the more important
implementation issues are sorted out. The subsequent semantic formalization
of the language inevitably becomes a major task, and the complexity, volume,
and inscrutability of the result may constrain its usefulness. A classic
example of such an undertaking is the description of PL/1 in the Vienna
Definition Language[24].

An alternative technique of language design, exemplified to some extent in
LISP[26] and its recent derivatives, involves the specification of the
pragmatics of a language after decisions on some particular concise semantics
have been made. Unfortunately languages so designed tend to have serious
defects from a practical point of view and are abandoned or complicated by the
addition of ad hoc mechanisms to make them more useful.

The designer of a language is thus confronted with a choice between concise semantics and practical usability, and he justifiably tends to opt for the latter alternative. The extent to which semantic considerations may be reconciled with practical issues remains an important open question, and the development of practical languages with concise, elegant semantics is the long term goal of much of Computer Science research. The problem is being attacked from two discernible directions: (i) semantic formalisms which deal with the mechanisms of extant practical languages, such as the analysis of uninterpreted schemata[9,8,13,17,25]; and (ii) the adaptation of existing formalisms to very simple model languages such as the lambda calculus[2,3,5,15,22]. The work reported here falls naturally into the second category.

## 1.2: Applicative Languages

Familiar concepts of mathematics provide an informal semantics for many aspects of computer languages. Manuals for most programming languages relate various program constructs to such notions as real numbers, arithmetic, and functions, with which the reader is presumed to be acquainted. Often terminology and notation are borrowed from mathematics, implying some informal relation between, say, a FORTRAN "function" and the common mathematical notion of function. This relation is only approximate, since for example no mathematical analog has been established for the FORTRAN function which prints its argument on the teletype. In order to formalize the relationship between program constructs and mathematical notions, then, we focus our attention on the highly restricted class of applicative languages.

The semantic bases of applicative languages are the theories of mathematical functions, and the constructs of these languages are restricted to simple analogs of the related mathematical notions. Each applicative language provides a syntactic formalism for the representation of functions and their application to arguments, and the semantics of an applicative language is in general a rule for the association of expressions, constructed according to this formalism, with values from an abstract semantic domain containing functions and constants. Formalizing a consistent semantics for an

applicative language appears to be an easy first step in pursuing the general
problem of programming language semantics;   since set theory provides
satisfactory semantic domains, all that remains is the seemingly simple
association of expressions with set theoretic functions and constants.

Yet even this simple problem is plagued with complications, and it is only in
recent years that progress has been made in this area largely due to
techniques developed by Dana Scott[5,6,22].  In fact, the usual set theoretic
characterization of functions is not so well adapted to the semantics of
applicative languages as one might suspect:   type restrictions, placed on set
theoretic functions in order to avoid Russel's Paradox, are difficult to
reconcile with the natural proclivity of applicative languages for the
self-application of functions.  The work of Scott justifies our optimism that
such problems are tractable, and that the semantics of applicative languages
may be based on the mathematics of functions.  The extension of the resulting
semantics to non-applicative mechanisms such as assignment and side effects
however, remains an area of grave uncertainty, and it seems likely that
theories of functions will ultimately prove to be inadequate bases for the
semantics of programming languages in general.  In the meantime, however,
applicative languages and their functional semantic domains are probably the
closest we have come to a successful programming language semantics, and we
feel that there is much insight to be gained from further exploration of this
area.

The semantics of an applicative language L, then, may be viewed as a mapping
between the set of valid expressions in L (the _domain of discourse_ of L) and
and a semantic domain of abstract functions and constants.  A consequence of
the Turing Universality of L is that this mapping must be many to one; each
abstract semantic element has, in general, infinitely many representations in
the language L.  The semantic mapping thus leads naturally to a notion of
semantic equivalence between expressions in L, partitioning the domain of
discourse of L into equivalence classes each of which corresponds to a single
abstract semantic element.

## 1.3: The Thesis: Statement of the Problem

The problem which this thesis addresses is the characterization of the
_expressive power_ of an applicative language in terms of the structure of its
abstract semantic domain.  This process generally involves relating specific
applicative language features to the expressibility of particular classes of
functions, e.g. the solution of the FUNARG problem to the expressibility of
functions mapping integers onto an infinite range of semantically distinct
functions.

This work focuses on a very few specific language mechanisms, with particular
attention given to an applicative analog of multiprocessing.  Partial answers
are provided to such questions as:

1) Are there functions whose computability depends fundamentally on a notion
   analogous to multiprocessing?

2) What applicative mechanisms are necessary for the expression of such
   functions, and is the impact of these mechanisms on the structure of the
   semantic domain?

3) What is such relationship between such multiprocessing constructs and
   other issues of applicative language evaluation, such as evaluation
   order?

The work presented here might be characterized as a search for an applicative
language L which is _functionally complete_ in the sense that every computable
function definable on the semantic domain of L is expressible in L -- our
reluctance to cite this as the principal goal of the thesis is probably due to
our failure to find such a language.

## 1.4: Outline of the Thesis

The organization of the remaining chapters is as follows:

  Chapter 2 develops the basic framework through the presentation of three
      interpreters for applicative languages, designated S (stack environment),
      T (tree environment), and N (normal order).  Each interpreter exemplifies

a typical language limitation and each is used to relate a specific
language characteristic to the expressibility of a particular class of
functions.

Chapter 3 demonstrates a particular computable function which is
inexpressible both in N and in the lambda calculus, and relates this
inexpressibility to the semantic requirement that an expression in these
languages have at most a single value. Two alternative language
extensions are discussed, each of which solves this specific
expressibility problem. The solutions involve, respectively, primitives
for coding the representation of functions as integers and a
multiprocessing primitive called EITHER. Each of these extensions
requires modification of the structure of the semantic domain, with the
use of coding leading to drastic and undesirable consequences. For this
and related reasons, EITHER is chosen. To account for the semantics of
EITHER, the semantic domain of N is expanded into a power set and each
expression X is associated semantically with an enumerable set containing
the admissible values of X.

The formalization of EITHER-augmented languages may procede in several ways,
differing in the restrictions placed on evaluation order. Chapters 4, 5, 6,
and 7 deal with certain formal theories, based on the lambda calculus, for the
reduction of expressions involving the EITHER construct:

Chapter 4 provides basic definitions and presents the Either-R Theory, in
which lambda conversion is allowed only in expressions whose arguments
are in normal form. This restriction is motivated by the intuitive
desire to maintain the distributivity of functions over terms of an
EITHER clause, but it limits the power of languages based on this theory.

Chapter 5 develops a theory of *-conversion, designed to mitigate the
limitations imposed by the restricted lambda conversion of the Either-R
Theory. The element * is introduced as a canonical representation of
every nonterminating computation, and a syntactic mechanism is provided
for the reduction of expressions to approximations which are in normal
form. The use of *-conversion provides techniques for proving certain
relationships in the conventional lambda calculus. This chapter presents

results which are of interest independently of their relation to the development of the Either theories.

Chapter 6 presents the Either-R-* theory, combining the EITHER mechanism with *-conversion, and establishes its consistency. While this system retains the restriction on lambda conversion, it has the power of the lambda calculus augmented by the EITHER primitive. Thus, languages based on Either-R-* solve the specific expressibility problem raised in Chapter 3. Interpreters and semantics for such languages are discussed.

Chapter 7 presents the Either-K theory, which combines the EITHER construct with unrestricted lambda conversion. Significant semantic differences between the Either-R and Either-K theories are noted, and it is informally observed that the removal of the restriction on lambda conversion leads to the expressibility of certain functions which are inexpressible in the Either-R-* languages.

The last chapter summarizes the results of this work and proposes avenues for future research.


1.5: Functional Domains

An underlying assumption of this research is that the fundamental semantic intent of applicative languages is to provide computational models of mathematical functions. As a consequence of this assumption, we are inclined to view functions in an applicative language as approximations or models of abstract mathematical functions, and to treat any disparity between the behavior of the computational model and the corresponding mathematical function as a "bug" or idiosyncrasy in the language.

The thrust of this research is aimed at the limitations of particular applicative languages as models of systems of mathematical functions. We begin by specifying, in the next section, criteria which must be obeyed by applicative functions to be intuitively satisfactory as models of mathematical functions, and then distinguish for each applicative language L that subset of the domain of L containing only such intuitively satisfying functions. We call such a subdomain of L a _functional domain_ of L.

## 1.5.1: Intuitive Criteria for Functions

Restricting our attention for the moment to unary (single argument) functions, we note that

1) A function $f$ is a **mapping** from a domain $D_f$ to a range $R_f$. The set-theoretic model of $f$ is a set of ordered pairs, $\{...\langle D_i,R_i\rangle...\}$, such that $f[D_i]=R_i$ if and only if $\langle D_i,R_i\rangle$ is an element of $f$.

2) A function $f$ may be **partial** over domain D, i.e., there may be elements $D_i$ in D such that $f[D_i]$ is undefined; this corresponds to the practical situation of a nonterminating computation or a computation which results in an error condition. We shall refer to such a computation as **divergent**.

3) If $f$ and $g$ provide the same mapping, then they are the same function.

4) $g$ is a **subset** of $f$ (in the set-theoretic sense) if and only if for every $D_i$ in the domain of $g$, $g[D_i]=R_i$ implies $f[D_i]=R_i$.

Given a language L and a function $f$, a principal intuitive requirement is the distinction between the function $f$ and the various algorithms (or expressions in L) which may be used to compute $f$. A major complication in the semantics of applicative languages arises from this many-to-one correspondance between algorithms and functions, particularly in light of the well known undecidability of equivalences between algorithms.

## 1.5.2: Functional Domain: Definition

The intuitive considerations of the previous section motivate the following definition:

Defn 1.1:   A **functional domain** F is a set containing the set N of natural numbers and computable functions,[1] along with an equivalence relation ~ such that:

---

[1] Unless specifically stated, we shall use the term **function** with no implied type restrictions. Thus functions include functionals of arbitrary order, consistent with the typeless character of the applicative languages considered here.

1) if $\underline{x}$ is in N or $\underline{y}$ is in N, then x~y if and only if x=y.

2) if neither $\underline{x}$ nor $\underline{y}$ is in N, then x~y if and only if for every $\underline{z}$ in F, $\underline{x}[\underline{z}]$~$\underline{y}[\underline{z}]$ or both diverge together.

3) if $\underline{x}$~$\underline{y}$, then for every $\underline{z}$ in F, $\underline{z}[\underline{x}]$~$\underline{z}[\underline{y}]$ or both diverge together.

Clause (1) simply asserts that different numbers, eg 2 and 3, are semantically different objects.  Clause (2) asserts that any object in F that is not a number is a function, and moreover that functions are semantically equivalent if and only if they perform equivalent computations for every set of arguments.  Clause (3) insists that the application of a function to semantically equivalent arguments yield semantically equivalent values.

An expression $\underline{z}$ is said to be _functional_ over the domain F if, for every choice of $\underline{x}$ and $\underline{y}$ in F, $\underline{x}$~$\underline{y}$ implies that $\underline{z}[\underline{x}]$~$\underline{z}[\underline{y}]$ or both computations diverge together.  Thus (3) is the requirement that every function in a functional domain F be functional over F.

We note that the equivalence relation ~ is not, in general, computable. Furthermore, there may be elements $\underline{x}$ and $\underline{y}$ in F such that $\underline{x}$~$\underline{y}$ is not defined, that is, such that neither $\underline{x}$~$\underline{y}$ nor ~$(\underline{x}$~$\underline{y})$ is derivable from the above definition.

This definition is rather more specific than necessary.  The choice of natural numbers as a basis of semantically distinct constants, rather than, say, character strings or floating point numbers, is arbitrary.  In dealing with the lambda calculus we could make the apparently stronger requirement that _normal form_ expressions be semantically distinct, rather than just the particular normal form expressions which are numeric constants;  however it happens that the two alternatives are entirely equivalent in the context of our model languages, and our present definition is the less dependent on particular syntactic considerations.

## Chapter 2:
## Interpreter Structure and Expressive Power

In this chapter several illustrative interpreters for applicative languages
are presented, and compromises in their implementation are related to the
inexpressibility of certain functions. The model interpreters are taken from
Dertouzos[3] where they are discussed and motivated in greater detail.


### 2.1: Syntax of Models

The essential components of an applicative language syntax are conventions for
the representation and application of functions. Typical applicative
languages provide for the representation of functions by either or both of the
following means:

1) A set of reserved symbols designating primitive functions whose semantics
   are basic to the language;

2) A convention for functional abstraction, or the definition of new
   functions by means of expressions containing variables.

The pure lambda calculus of Church[1] is illustrative of languages using only
the abstraction mechanism; the combinatory calculus of Curry[12] exemplifies
the use of primitives without abstraction. Curry[12] has demonstrated the
equivalence of these mechanisms, with minor qualifications, and the choice
between them for our purposes is largely a matter of convenience; we provide
here syntactic constructs for both.

Beyond these constraints, the syntactic details of the languages discussed
here are not important. A LISP-like syntax has been chosen for the
development of the models and to provide a definite basis for examples and
illustrations, although the results and examples may be translated to conform
to other syntactic conventions which are consistent with these constraints.
Syntactic characteristics of our model languages include:

1) A finite alphabet including the alphanumeric characters and the special
   characters "(" and ")";

2) A countably infinite set of <u>identifiers</u>, each a finite string of alphanumeric characters of which the first is alphabetic;

3) A set of numeric constants, each represented in the language by a finite string of digits.

The elements of the model applicative languages are the <u>applicative expressions</u> (<u>AE</u>'s) whose syntax is given by:

| | |
|---|---|
| \<AE\> | := \<identifier\> \| \<number\> \| \<combination\> \| \<lambda expression\> |
| \<identifier\> | := \<letter\> \| \<identifier\>\<digit\> \|\<identifier\> \<letter\> |
| \<combination\> | := ( \<AE list\> ) |
| \<AE list\> | := \<AE\> \| \<AE\> \<space\> \<AE list\> |
| \<lambda expression\> | := ( LAMBDA (\<bvl\>) \<AE\>) |
| \<bvl\> | := \<null\> \| \<identifier\> \<space\> \<bvl\> |
| \<number\> | := \<digit\> \| \<digit\> \<number\> |
| \<letter\> | := A \| B \| ... \| Z |
| \<digit\> | := 1 \| 2 \| ... \| 0 |

We assume of these model languages that data is either numeric or functional, that is, that the <u>value</u> computed for any applicative expression must be either a natural number or a function.[1]  An expression X is <u>atomic</u> if X is an identifier or a number; in addition the following syntactic forms have special meaning in our model languages:

1) The syntactic form of a lambda expression is

$$(LAMBDA(a_1 \ a_2 \ ... \ a_n) \ b)$$

---

[1] Our decision to ignore for the present other common data types (floating point numbers, arrays, character strings, lists) is justified by their codability as numbers, so that our results concerning processing of numeric data may be extended to the processing of these other data as well.

where LAMBDA is a reserved identifier in the language, the $a_i$ are identifiers on the bound variable list of the lambda expressions, and the expression b is the body of the lambda expression.

2) The syntactic form of the application of the procedure (function) f to arguments $x_1 \ldots x_n$ is

$$(f\ x_1\ \ldots\ x_n)$$

Here f is presumed to be the representation of a functional datum, and the $x_i$ are representations of arbitrary data which are supplied to the function f as arguments.

There is in each language a small finite set of reserved identifiers used to denote primitive functions. Our initial models will include the following primitive function identifiers:

1) The logic values T and F, primitive functions defined such that the value of the application

$$(T\ a\ b)$$

is the value of the expression a, regardless of whether the value of the expression b is defined. Similarly, the value of

$$(F\ a\ b)$$

is the value of the expression b whether or not a has a value.

2) The function PLUS of 2 arguments, defined such that the value of the expression

$$(PLUS\ a\ b)$$

is the sum of the values of the expressions a and b. The value of the application of PLUS is undefined if either of the values of a or b is nonnumeric.

3) The function GREATER of 2 arguments, defined such that the value of the expression

$$(GREATER\ a\ b)$$

is the primitive function T if $\underline{a}$ has a higher numeric value than the
expression $\underline{b}$, and F if the value of $\underline{a}$ is less than or equal to the value
of $\underline{b}$.

We shall often refer to an identifier which is not a primitive function symbol
as a __variable__.  An occurrence of the variable $\underline{y}$ in the expression X will be
termed a __free occurrence__ if one of the following applies:

   1) X is identically the variable y; or

   2) X is of the form $(A_1 \ldots A_n)$ and the occurrence of y is free in one of
      the $A_i$; or

   3) X is of the form $(LAMBDA(a_1 \ldots a_j)M)$, y does not occur in the bound
      variable list $(a_1 \ldots a_j)$, and the occurrence of y is free in M.

An occurrence of the variable y which is not free is __bound__.

## 2.2:  Curried Functions

The syntactic provision made here for functions of multiple arguments requires
certain further elaboration.  We may reasonably demand, for example, the
ability to express the function MPLUS defined such that the value of (MPLUS m)
is the m-ary function which returns the sum of its m arguments.  Such
functions are, in general, unrepresentable unless some primitive mechanism is
provided within the language for the abstraction of multiple argument
functions.  We might consider the abstraction primitive ALPHA, defined such
that the value (ALPHA F G m) is the m-ary lambda expression

$$(LAMBDA(X_1 \ldots X_m)(G \; X_m \; (F \; X_1 \; \ldots \; X_{m-1})))$$

where F and G are presumed to represent (m-1)-ary and binary functions,
respectively.  We might then define MPLUS so that (MPLUS 2) returns PLUS, and
(MPLUS n) returns (ALPHA (MPLUS n-1) PLUS n) for n>2.

Such a primitive is, however, unnecessary in most languages.  The technique of
Curried functions[1] may be used to couch multiple-argument functions in terms

---

[1] named in honor of H.B. Curry who developed this technique; see [12]

of unary functions, whence the application of F to arguments $A_1$ $A_2$ ...$A_n$
becomes

$$( \ldots ((F\ A_1)\ A_2) \ldots A_n)$$

and the n-ary lambda expression (LAMBDA($A_1$ $A_2$...$A_n$)M) becomes

    (LAMBDA($A_1$)
        (LAMBDA($A_2$)
            ---
            (LAMBDA($A_n$)M) ... ))

The convention of Curried functions simplifies the presentation of proofs and
interpreters, as only single argument functions need be considered;  we
therefore hastily adopt it for our present purposes.  The conventional
multiple argument syntax is slightly less complicated, however, and tends to
greater clarity than the use of Curried functions; we consequently allow
ourselves the informality of switching freely between the two conventions at
our convenience.  We may then consider instances of the multiple argument
syntax as an abbreviation for the corresponding Curried syntax, which we take
as basic.

An exception must be made in the first model language presented, however, as
the FUNARG problem does not interact gracefully with Curried functions; hence
in this case the assumption of single argument functions is not made.


2.3:  The FUNARG Problem


We are now in a position to give an example of a functionally incomplete
language, which we call S.  S is an abstraction of the applicative subset of
LISP and similar stack-oriented languages; it serves to introduce the notion
of _environment_, and demonstrates that certain minimal structural constraints
on environment handling mechanisms are necessary for the expressibility of a
particular class of functions.

## 2.3.1: The S model

An <u>environment</u> is a linear sequence of ordered pairs (or <u>bindings</u>) (x,v), where x is an identifier and v is a value. Environments are thus a mechanism for the use of identifiers as variables, serving to record the values associated with each variable. We represent the environment which binds the variable $X_1$ to the value $V_1$, $X_2$ to $V_2$, and so on, as

$$((X_1,V_1)(X_2,V_2) \ldots )$$

The environment structure of the interpreter for S may be viewed as a stack, bindings being pushed onto the environment from the left at the start of the application of a lambda expression, and subsequently being popped from the environment at the completion of that application. The S interpreter finds the current value for a variable X by looking, in turn, at each binding starting with the leftmost; when a binding whose first element is X is encountered, the associated value (the second element of the binding) is taken as the value of X. We may describe this operation by defining a primitive function <u>lookup</u> of two arguments, corresponding respectively to the identifier to be evaluated and the environment in which its value is to be found:

```
lookup[x;((X_1,V_1)(X_2,V_2)...(X_n,V_n))]=
        if x=X_1 then V_1;
        else lookup[x;((X_2,V_2)...(X_n,V_n))]
```

We now describe the interpreter for S as a function defined recursively as follows:

```
S[x;e] =
            if x is a number, then x;
            if x is a member of {T,F,GREATER,PLUS} then x;
            if x is an identifier then lookup[x,e];
            if x is a lambda expression then x;
            if x is of the form (T y z) then S[y;e];
            if x is of the form (F y z) then S[z;e];
            if x is of the form (GREATER y z) then:
                if S[y;e]>S[z;e] then T;
                else F;
```

if x is of the form (PLUS y z) then $S[y;e]+S[z;e]$;

if x is of the form $((\text{LAMBDA}(s_1...s_n)\ b)\ y_1...y_n)$ where the $s_i$ are identifiers, then
$S[b;(s_1,S[y_1;e])...(s_n,S[y_n;e])e]$;

if x is of the form $(y\ z_1\ z_2\ ...\ z_n)$ where y is not a lambda expression, then $S[(S[y;e]\ z_1\ ...\ z_n);e]$;

else undefined

Thus $S[x;e]$ computes the value of the expression x in the environment e. $S[x;\emptyset]$ (where $\emptyset$ is the empty environment) computes the value of x on an S evaluator in its initial "bare" state;  we may refer to this simpley as the S value of x.


## 2.3.2:  Arithmetic Completeness of S


We refer to a language as arithmetically complete if every computable first order[1] function is representable as a procedure of that language. We show that S is arithmetically complete by showing that for every first order partial recursive (hence computable) function there is a corresponding function in S.  The constructions of this section are adaptations of those appearing in Dertouzos[3] and are included here primarily for sake of illustration; while each subsequent model language is also arithmetically complete, similar constructions apply in each case and will not be repeated.

As a preliminary step, we consider the S function given by:

```
(LAMBDA(X Y)
    ((LAMBDA(X Y D)(D X Y)) X Y
        (LAMBDA(X Y)((GREATER X Y)
            (PLUS 1 (D X (PLUS 1 Y)))
            0)) ))
```

which computes the "recursive difference" function

---

[1] Following the terminology of logic, a first order function contains only numbers in its range and domain, and functions of order j may contain (in addition to numbers) functions of order less than j.

D[x;y] = if x>y then x-y else 0;

by the algorithm

D[x;y] = if x>y then 1+D[x;y+1];
                      else 0;

Note that the extra two layers of LAMBDA binding serve only to bind the free occurrence of the identifier D within its own definition, and thus to make the recursive function operate properly on S.[1]

We may define the predecessor function

P[x] = if x<1 then 0 else x-1;

in S by the expression:

(LAMBDA(X)(D X 1))

where D is the recursive difference function defined above.

Now we shall demonstrate that every partial recursive function of first order is representable as a function in S. In the following, lower case letters represent partial recursive functions while upper case letters denote their corresponding S functions:

1) For every pair of natural numbers n and m, the m-argument constant function of value n is expressed in S as:

$$(LAMBDA(X_1 \ldots X_m) \; n)$$

2) For every pair of numbers n and m, the m-ary projection function which returns the value of its nth argument is expressed in S by:

$$(LAMBDA(X_1 \ldots X_m) \; X_n)$$

3) The successor function is expressed in S by:

$$(LAMBDA(X)(PLUS \; 1 \; X))$$

---

[1] This is one of several "tricks" which may be used to perform recursion on S. The necessity of such tricks stems from the expressive inadequacy of S: the paradoxical combinator, Y, is not expressible as a function of S. For discussion of Y, see Rosenbloom[4] and Landin[2]; for a general discussion of recursion on S see Dertouzos[3].

4) (<u>composition</u>) For every choice of numbers n and m, m-ary partial
recursive functions $g_1 \ldots g_n$, and n-ary function f, the m-ary function h
defined by

$$h[x_1;x_2;\ldots;x_m] = f[g_1[x_1 \ldots x_m], \ldots ,g_n[x_1 \ldots x_m]]$$

is expressed in S as

       (LAMBDA($X_1$ ... $X_m$)(F
                      ($G_1$ $X_1$ ... $X_m$)
                      ... ($G_n$ $X_1$ ... $X_m$) ))

where F, $G_1 \ldots G_n$ are the S expressions corresponding to f and $g_1 \ldots g_n$,
respectively.

5) (<u>primitive recursion</u>) If the n-ary partial recursive function g and the
(n+2)-ary primitive recursive function f are expressible in S as G and F,
respectively, then the (n+1)-ary function h defined by:

$$h[x_1,\ldots x_n,0] = g[x_1,\ldots x_n]$$
$$h[x_1,\ldots,x_n,y+1] = f[x_1,\ldots,x_n,y,h[x_1,\ldots,x_n,y]]$$

may be expressed in S by

       (LAMBDA($X_1$ ... $X_n$ Y)
            ((LAMBDA($X_1$ ... $X_n$ Y H)(H $X_1$ ... $X_n$ Y)) $X_1$ ... $X_n$ Y
                (LAMBDA($X_1$ ... $X_n$ Y)((GREATER Y 0)
                     (F $X_1$ ... $X_n$ (P Y) (H $X_1$ ... $X_n$ (P Y)))
                     (G $X_1$ ... $X_n$) )) ))

where P is the representation of the predecessor function given earlier.

6) (<u>mu-recursion</u>) If the (n+1)-ary <u>total</u> recursive function h is expressible
in S by H, then the partial recursive function g defined by

       $g[x_1;\ldots;x_n]$ = the <u>least</u> y for which
                       $h[x_1;\ldots x_n;y] = 0$

is represented in S by

```
(LAMBDA(X  ... X ))
        1       n
        ((LAMBDA(R)(R 0))
                (LAMBDA(Y)((GREATER (H X  ...X  Y) 0)
                                        1      n
                        (R (PLUS 1 Y))
                        Y)) ))
```

Finally, we note that the class of recursive functions is by definition
exactly that class of functions obtainable through finitely many applications
of the above six rules; hence the S representations given in the rules
constitute a technique for constructing an S expression which represents any
function which can be shown to be partial recursive.


2.3.3:  Functional Incompleteness of S

Recall that the functional completeness of a language L requires that every
computable function defined on the semantic domain of L be expressible in L.
Since the natural numbers and (by the preceding section) first order functions
are included in the semantic domain of S, every second order function is
definable on the domain of S.  The functional incompleteness of S may then be
demonstrated by showing that a simple second order function is not expressible
as an S function.  We begin by observing that some higher order functions are
expressible in S, e.g. the function g (the "twice" function) given by

$$g[f;x] = f[f[x]]$$

is expressible in S as

    (LAMBDA(F X)(F (F X)))

hence it cannot be argued that only first order functions are expressible in
S.  The weakness in S which we will demonstrate involves the inexpressibility
of certain second order functions, notably functions which contain free
variables and which appear as arguments or values (i.e., bodies) of lambda
expressions: the so called FUNARG problem.[1]

---

[1] General awareness of the FUNARG problem (as well as its name) arose from
early experience with LISP.  For discussion see Weizenbaum[23], Moses[10] or
Dertouzos[3].

Consider the unary function f, whose domain contains only integers and whose range contains only first order functions, defined by

$$f[x] = \text{that function } g \text{ defined by}$$
$$g[y] = x+y$$

The function f is computable; it may in fact be expressed in the lambda calculus by

(LAMBDA(X)(LAMBDA(Y)(PLUS X Y)))

To show that f is not expressible in the language of S, the following definition is useful:

Defn 2.1:   We say that the expression a appears as a subexpression of the expression b if any of the following are true:

1) The expressions a and b are identical;

2) b is of the form

$$(b_1 \ b_2 \ \ldots \ b_n)$$

where a appears as a subexpression of one or more of the $b_i$;

3) b is of the form

$$(\text{LAMBDA}(X_1 \ \ldots X_n)B)$$

where a appears as a subexpression of B.

We say informally that b contains a if a appears as a subexpression of b.

The basis of the inexpressibility of f in S is established by the proof of

Lemma 2.2:   Let A be any applicative expression and let B be a lambda expression appearing neither as a subexpression of A nor in the environment e.   Then B does not appear as a subexpression of S[A;e].

proof is by induction on the recursion depth of S[A;e].

basis For the following syntactic classes of A, the computation of S[A;e] involves no recursion:

Case 1: A is a number, a primitive function identifier, or a lambda expression.   Then S[A;e]≡A, and the lemma is trivially satisfied as

B is not a subexpression of A.

Case 2: A is an identifier other than a primitive function symbol.  Then
       S[A;e] is lookup[A;e] which cannot contain B since by assumption the
       environment e does not contain B.

induction: The remaining cases of the syntax of A follow;  for these we
assume that the Lemma holds for recursive calls to S.

Case 3: A is an application of GREATER or PLUS; then the value of S[A;e]
       is a number or logic value and does not contain B.

Case 4: A is the application of a logic value T or F to arguments $A_1$ and
       $A_2$.  Neither $A_1$ nor $A_2$ can contain B since A does not contain B;
       hence the inductive hypothesis applies to either of the computations
       $S[A_1;e]$ and $S[A_2;e]$ and B cannot appear in S[A;e] which is one of
       these values.

Case 5: A is the application of a lambda expression $(LAMBDA(X_1...X_n)M)$ to
       the arguments $A_1...A_n$.  By the inductive hypothesis, B does not
       appear in any of the values $S[A_1;e]...S[A_n;e]$, hence the new
       environment $e' \equiv (X_1,S[A_1;e])...(X_n,S[A_n;e])e$ does not contain B.  As
       a subexpression of A, M cannot contain B; thus the inductive
       hypothesis applies to the value $S[M;e']$ returned as the value of
       S[A;e].

Case 6: A is the application of Y to the arguments $A_1...A_n$, where Y is
       neither a lambda expression nor a primitive function symbol.  Y is a
       subexpression of A and by assumption does not contain B as a
       subexpression.  Then the inductive hypothesis applies to the
       computation of $S[Y;e] \equiv Y'$, and $Y'$ does not contain B; a second
       application of the inductive hypothesis reveals that B cannot appear
       as a subexpression of $S[(Y' A_1...A_n);e] \equiv S[A;e]$.

These cases are exhaustive, completing the proof.


We can now characterize a major weakness of the language S by

Thm 2.3:  Every function expressible in S whose domain contains only numbers
          may have at most finitely many functions in its range.

   Proof: Functional values in S must be either primitive function identifiers
          or lambda expressions.  As there are finitely many primitive functions,
          we need only show that each function of numbers in S has finitely many
          lambda expressions in its range.  Implicit in this argument is the fact
          that the number of functions expressed by a set of lambda expressions is
          no greater than the number of lambda expressions in the set.  Each lambda
          expression which contains no nontrivial occurrences of free variables
          represents (though not necessarily uniquely) a single function; lambda
          expressions with nontrivial occurrences of free variables (i.e., which
          compute different functions in differing contexts) do not correspond
          semantically to functions.

          By lemma 2.2, a function of integers can have lambda expressions in its
          range only if they appear as subexpressions of the function, since for
          any integer $n$ and expression $f$ the expression $(f\ n)$ can contain the
          lambda expression $g$ as a subexpression only if $g$ is a subexpression of $f$.
          As the function must be represented by a finite expression in the
          language S, it may contain only finitely many lambda expressions as
          subexpressions and hence has finitely many lambda expressions in its
          range.

Clearly, the function $f$ defined at the beginning of this section is a function
of integers having infinitely many functions in its range;  we conclude that $f$
is not expressible in S.  The problem may be characterized as inadequate
handling by S of lambda expressions containing free variables.  It is apparent
that free variables are evaluated in the environment in which a function is
applied, rather than the environment in which it is evaluated.  Thus lambda
expressions with free variables have the property that the computation which
they perform depends on values in the environment of their caller; this
dependency constitutes an implicit input and justifies our exclusion of such
lambda expressions from the class of functions.  Yet proper S functions may
include such lambda expressions as subexpressions; witness the S function

(LAMBDA(X)((LAMBDA(Y)(PLUS X Y)) 3))

which contains no free variables and hence no implicit inputs.  The variable
X, however, appears free in the lambda expression in its body;  this innermost
lambda expression is not a function.  The question of the contribution of free
variables to the functional richness of S naturally arises at this point: Are
there functions which are expressible in S _only_ through the use of free
variables?  Our suspicions lead to the conjecture that every function _f_
expressible in S may be represented by an expression _F_ in which no lambda
expression appearing as a subexpression contains free occurrences of
variables.  This conjecture does not completely deny the usefulness of free
variables on the S machine.  Indeed, lambda expressions with free variables
are moderately well behaved when passed _downward_, i.e., as arguments to
functions;  under these circumstances, the principal danger is due to possible
conflicts with variables bound by the functions to which the lambda
expressions are passed.  They may, however, be considered to be "limited
functions" with the qualification that they be applied within the scope of the
free variables in their original environment and that they may not be passed
to functions whose bound variable list includes any of the free variables.
Such qualifications seriously impair the semantic clarity of the language
imposing them.


2.4:  Evaluation Order

The functional incompleteness of S was shown to be related to the specific way
in which S associates values with variables in an interpreted program: i.e.,
the environment structure of S.  The remaining sections of this chapter
present model interpreters with alternative environment structures, and which
solve the specific problem demonstrated in S;  however, they demonstrate
similar inadequacies in the organization of _control structures_, i.e. the data
structure specifying which computations are to be performed and their relative
sequence.[1]

---

   The notion of _control structure_ has never, to the author's knowledge, been
adequately formalized.  Informally it is the bookkeeping mechanism necessary
to resolve algorithms into sequences of operations -- e.g., the use of a stack
to record the return points of calls to a recursive subroutine.

The first model to be presented is T, similar to S except that its environment
is structurally a tree rather than a stack.  It is argued that T and S share a
deficiency which stems from their _evaluation order_, in particular, from their
uniform evaluation of arguments regardless of whether the resulting values are
essential to the computation.  T is thus functionally incomplete due to
evaluation order.

The N model, discussed in section 2.5, is closely related to the normal order
evaluation of the lambda calculus.  It is superior to T in that every
expression having a T value has an equivalent N value, while certain
expressions have N values but not T values.


## 2.4.1:  The T Model

The traditional solution of the environment problem of S involves a new
"internal" representation of a function, called a _closure_.  A closure
includes, in addition to the information in a lambda expression, a
specification of the environment in which its free variables are to be
evaluated.  As the closure mechanism may require the retention of environment
branches corresponding to functional applications from which control has been
returned, the environment becomes a _tree_ rather than the _linear stack_ of S;
hence we call our new language T.  The difference between T and S is that in
T, the lambda expression

$$(LAMBDA(s_1...s_n)\ b)$$

is no longer self evaluating.[1]  Its value, in environment $e$, is

$$(FUNARG(s_1...s_n)\ b\ e)$$

which is the representation of a _closure_ in T.  We define T as follows:

T[x;e] =

               if x is a number, then x;

               if x is a member of {T,F,GREATER,PLUS} then x;

               if x is an identifier then lookup[x;e];

---

[1] We say an expression X is self evaluating if the value of X is X.

if x is of the form (T y z) then $T[y;e]$;

if x is of the form (F y z) then $T[z;e]$;

if x is of the form (GREATER y z) then:

    if $T[y;e]>T[z;e]$ then T;

    else F;

if x is of the form (PLUS y z) then $T[y;e]+T[z;e]$;

if x is of the form

  $(LAMBDA(s_1...s_n)$ b) then

  $(FUNARG(s_1...s_n)$ b e);

if x is of the form

  $((FUNARG(s_1...s_n)$ b $e_1)$ $y_1...y_n)$ then

  $T[b;(s_1,Y[y_1;e])$ ... $(s_n,T[y_n;e])+e_1]$;

if x is of the form $(y\ z_1\ z_2\ ...\ z_n)$ where y is not a

FUNARG closure, then

  $T[(T[y;e]\ z_1\ ...\ z_n);e]$;

else undefined;

We note that a lambda expression is not applied directly; it is first converted to a closure (by its evaluation), and then applied by the evaluation of its body in an environment formed by appending the bindings of its bound variable list to the closure environment. Thus the free variables of a lambda expression are evaluated in the environment in which the lambda expression is evaluated. The reader may verify that the function represented in the lambda calculus by

$$(LAMBDA(X)(LAMBDA(Y)(PLUS\ X\ Y)))$$

which the preceding section showed to be inexpressible in S, is expressible in T (indeed, by the same lambda expression).

## 2.4.2: Functional Incompleteness of T

Except for the special cases involving the application of the primitives T and F, the T evaluator uniformly evaluates the expressions supplied to an operator as arguments before the operator is applied. This order of evaluation, which has been termed **applicative order**, has the virtue that each subexpression of

an AE is evaluated at most once, whereas in the normal order evaluation of the
lambda calculus an argument to a function may be evaluated many times.  The
disadvantage of applicative order evaluation is that arguments may be
evaluated (once) even though their value is irrelevant to the computation;
this is not merely a matter of occasional inefficiency, since the irrelevant
argument may not be defined whereby the entire computation diverges.  Consider
the case of the trinary projection function

$$P_{31}[x;y;z]=x$$

which returns its first argument regardless of whether its remaining arguments
have defined values.  The applicative-order counterpart of $P_{31}$ is represented
in T by the expression:

$$f_{31}=(LAMBDA(X\ Y\ Z)\ X)$$

This expression does not return a value under T-evaluation unless all three
arguments have defined values.

Our decision to distinguish between $P_{31}$ and $f_{31}$ in effect recognizes the
undefined element, *, as a member of the functional domains of our applicative
languages.  Intuitively, * represents the "value" of those computations which
do not terminate, and whose expressibility in each language L is guaranteed by
the Turing universality of L.

We now show that $P_{31}$ is not expressible in T:

Thm 2.4:  For every AE f, the T value of the expression

$$(f\ 3\ *\ *)\qquad\qquad [2.5]$$

(where * denotes any expression whose T value is undefined) is undefined.

proof:  We consider exhaustively the possible T values of the operator f:
If f is a number or a primitive operator, then the value of [2.5] is
undefined due to an error in functionality, i.e. the application of a
primitive to arguments for which it is not defined.  may assume that f is
either a combination or a lambda expression, in which cases the value of
the combination is the value of the application of the T value of f to
the specified arguments.  If the value of f is a number or a primitive,

[2.5] is again undefined due to an error in functionality. Hence the value of $f$ must be a closure. The computation of the application of a closure involves binding the values of each argument onto the environment, hence the evaluation of [2.5] entails evaluation of each argument. Since not every argument has a defined T value, the value of [2.5] is undefined.

Since clearly the projection $P_{31}$ has the property of $f$ in Theorem 2.4, T must be functionally incomplete if we are to consider $P_{31}$ a function.


## 2.5:  The N model

This section introduces an applicative language whose interpretation involves normal order evaluation. The superiority of N over T derives from this revised evaluation order of N, which permits an expression to be evaluated even though subexpressions of it may be undefined. A theorem of Church and Rosser establishes that if an AE, A, has a value under any evaluation order, then it has that value under normal order evaluation; thus in terms of evaluation order, N is optimal.

The simplest implementations of normal order evaluation involve the substitution of argument text in the bodies of lambda expressions, rather than the binding of argument values in environments. While the explication (and implementation) of such substitution algorithms is relatively straightforward, evaluation by simple substitution is often inefficient since

   1) It involves making many copies of program text during execution, and

   2) It often involves multiple evaluations of the same subexpression.

For reasons of efficiency, substitution evaluators are thus primarily of theoretical interest.

More efficient implementations of normal order evaluation retain the environment structure of the T model, and introduce additional mechanism to indicate which bound expressions have or have not been evaluated. Since the environment implementations of normal order evaluation involve considerable

bookkeeping machinery and are hence conceptually much more complex than the
substitution algorithms, they will not be pursued.


### 2.5.1:  Axioms for the Lambda Calculus

The primordial applicative language is the lambda calculus, which has been the
subject of much investigation since its conception by Alonzo Church in the
1930s.  The semantic basis of the lambda calculus is a set of axioms which
define an equivalence relation, =, on expressions of the language.  Each axiom
may be interpreted as a _conversion rule_ (or _reduction rule_) in the sense that
it provides a means for converting (or reducing) an AE to an equivalent (under
=) AE having a different form.  The presentation of the axioms in this chapter
is somewhat informal, serving primarily as motivation for the N interpreter;
the interested reader is referred to Curry[12] and Hindley[21] for further
detail.  Related issues are also covered in greater depth in later chapters of
this report.

The axioms of the lambda calculus are of 4 types, designated _alpha_
(equivalence under renaming), _beta_ (function application), _delta_ (primitive
function definition), and, in some formulations, _eta_.  The delta and eta
axioms are not used in all formulations.  The eta axiom seems to serve no
important function in the evaluation of expressions and will be presented here
only in passing.  The delta axioms may be avoided by well known coding
techniques which involve the representation of nonfunctional data, e.g.
natural numbers, as lambda expressions.[1]

The formulation which will be primarily referred to in subsequent chapters
comprises the alpha, beta, and delta axioms, and is often termed the
beta-_delta_-calculus in the literature.  Unless otherwise qualified, generic
references to "the lambda calculus" in this report denote the beta-delta
calculus.

The equivalence relation = of interconvertability is generated by a relation

---

[1] Many such codings are possible; a popular choice represents 0 by the
expression (LAMBDA(X)(LAMBDA(Y)Y)) and the number n+1 by
(LAMBDA(X)(LAMBDA(Y)((N X)(X Y)))) where N is the representation of the number
n.  For development of such a coding, see Church[1].

-> of <u>reducibility</u>; hence X->Y implies X=Y which, in turn, implies Y=X. Reducibility is in general antisymmetric, however; thus -> provides an ordering of equivalent expressions which has important ramifications in the lambda calculus. The relation -> is defined to be a <u>monotone relation</u>[1] meaning that it has the following properties:

Reflexivity: For every X, X->X;

Transitivity: If X->Y and Y->Z, then X->Z;

Monotonicity: If X->Y and B is the result of substituting, in an expression A, X for an occurrence of Y, then B->A.

The relation = is in addition an equivalence relation; hence X=Y implies Y=X.

Central to the axioms is the <u>substitution rule</u>, S, of fundamental importance to the lambda calculus as well as the theories of the following chapters of this report. S is formulated as a three argument function, such that the meaning of S[X;Y;Z] is roughly "the result of substituting the expression X for free occurrences of the variable Y in the expression Z. The definition of S is further complicated, however, by the requirement that the operation S[X;Y;Z] not introduce conflicts between free variables in the expression X and bindings of X within Z. There is a long history of incorrect algoritms for S; the definition given here is due to Curry:

Defn 2.6: For expressions X and Z, and variable Y, the expression S[X;Y;Z] is defined as follows:

1) If Z=Y, then X;

2) If Z is a primitive, number, or identifier other than Y, then Z;

3) If Z is of the form $(Z_1 Z_2)$ then $(S[X;Y;Z_1]\ S[X;Y;Z_2])$;

4) If Z is of the form (LAMBDA(A)M) where Y=A, then Z;

5) If Z is of the form (LAMBDA(A)M) where Y is different from A, then (LAMBDA(B)S[X;Y;S[B;A;M]]). where the variable B is chosen as follows:

   i) If Y does not occur free in M or if A is not free in X, then B=A;

   ii) Else B is any variable which occurs free neither in M nor in X.

---

[1] Terminology after Curry[12]

We now procede to the statement of the axioms:

Axiom _alpha_: If E is a lambda expression of the form (LAMBDA(X)M) and the
    variable Y does not occur free in M, then E->(LAMBDA(Y)S[Y;X;M]).

We say that expressions A and B are _congruent_ if A can be converted to B by
alpha conversion alone.  Note that if X->Y by alpha conversion then Y->X by
alpha conversion;  hence X=Y.  Congruence is thus symmetric and transitive,
and under most circumstances congruent expressions may be treated as
identical.  We say that expression X is in _normal form_ if the only reduction
which can be performed on X is alpha conversion.[1]

Axiom _beta_: If E is an expression of the form ((LAMBDA(X)M) A) then
    E->S[A;X;M].

Axiom _eta_: If E is an expression of the form (LAMBDA(X)(M X)) where X does not
    appear free in M and M is a lambda expression, then E->M.

Axiom _delta_: If E is an expression of the form (F $A_1$ $A_2$ ... $A_n$) where F is a
    primitive function symbol and each Ai is in normal form and contains no
    free variables, then E->f[$A_1$;...;$A_n$] where f is the operation denoted by
    F.

The following two theorems are of fundamental importance in the lambda
calculus.  The first is due, in its initial primitive form, to Church and
Rosser and is referred to in the literature as the Church-Rosser Theorem:

Thm 2.7:  Let X and Y be expressions such that X=Y.  Then there exists an
    expresion, Z, such that X->Z and Y->Z.

_proof_ may be found in Curry[12] or Hindley[21] and elsewhere.

The Church-Rosser Theorem shows that the lambda calculus is consistent in the
sense that the relation = is nontrivial;  in particular, X=Y is not true for
incongruent expressions X and Y in normal form.  We can thus prove that
expressions X and Y are not interconvertible by finding normal forms X´ and

---

[1] This definition is recast more formally in the terminology of Chapter 4.

Y´, where X->X´ and Y->Y´, which are incongruent.

Unfortunately, not every expression X is convertable to an expression X´ in normal form. For example, the important expression

$$Y \equiv (LAMBDA(F)((LAMBDA(H)(F (H H)))(LAMBDA(H)(F (H H)))))$$

which is the "paradoxical combinator" of Curry, has no normal form. Further discussion in this area follows in Chapters 4 and 5, along with related technical developments.

A second important theorem, due to Corrado Boehm, has been proved only for systems which prohibit delta conversions:

Thm 2.8: Let X and Y be incongruent expressions in normal form, and let C and D be arbitrary expressions. Then there exists and expression Z such that C=(Z X) and D=(Z Y).

proof originally appeared in Boehm[20], in Italian; a proof in English appears in Curry[27].

Boehm´s Theorem guarantees that incongruent normal forms in the beta-eta calculus[1] are semantically distinct; in particular, the axiomatic assertion that any two incongruent normal forms are interconvertable results in an inconsistency. The extension of Boehm´s Theorem to systems which include delta conversions requires that the constants added to the pure lambda calculus also be semantically distinct. We might, for example, formulate a calculus including the numeric constants without providing any means for distinguishing between them: we could provide the primitive PLUS but not GREATER. While this formulation is valid in terms of the lambda calculus, Boehm´s Theorem is clearly inapplicable since there is no expression Z which distinguishes, say, between the normal forms 2 and 3.

---

[1] i.e., that formulation including axioms alpha, beta, and eta, but excluding delta conversions.

## 2.5.2: Normal order: Substitution

Each of the lambda calculus axioms provides a means by which an applicative
expression E may be reduced to an equivalent expression E´. While the axioms
themselves place certain restrictions on the order in which such reductions
may be performed,[1] the evaluator of an applicative expression has a great deal
of freedom to choose the order in which to evaluate subexpressions.
<u>Normal order</u> evaluation specifies that at each evaluation stage, the <u>leftmost</u>
reducible subexpression is to be converted.


### 2.5.2.1: The N Evaluator

We define the N value of an AE <u>x</u> as follows:

$N[x]$ =

> if x is a number, then x;
>
> if x is a member of {PLUS,GREATER} then x;
>
> if x is a lambda expression, then x;
>
> if x is of the form (PLUS <u>a</u> <u>b</u>) where $N[a]$ and $N[b]$ are
> both defined and numeric, then $N[a]+N[b]$;
>
> if x is of the form (GREATER <u>a</u> <u>b</u>) where $N[a]$ and $N[b]$
> are both defined and numeric, then if $N[a]>N[b]$ then
> (LAMBDA(X Y)X) else (LAMBDA(X Y)Y);
>
> if x is of the form ((LAMBDA(<u>a</u>)<u>b</u>)<u>c</u>) where <u>a</u> is an
> identifier and <u>b</u> and <u>c</u> are AE´s, then $N[b´]$ where b´
> is the result of substituting <u>c</u> for each free
> occurrence of <u>a</u> in <u>b</u>;
>
> if x is of the form (<u>a</u> <u>b</u>) where <u>a</u> and <u>b</u> are AE´s and <u>a</u>
> is not a lambda expression, then $N[(N[a] b)]$;
>
> else undefined;

Note that we have eliminated the primitives T and F, which are entirely
equivalent in N to the lambda expressions which replace them as values of
GREATER.

---

[1] Not every expression E containing applications of lambda expressions, for
example, is beta-reducible. Applications of axiom alpha, ie the renaming of
variables, may be required <u>before</u> axiom beta is applicable.

2.5.2.2:  Axiomatic Consistency of N

We show in this section that N evaluation is consistent with the semantics of
the lambda calculus by demonstrating that N preserves the equivalence relation
"=":


Thm 2.9:  Let E be any AE such that N[E] is defined.  Then E->N[E] where -> is
          the reducibility relation defined by the lambda calculus axioms.

    proof:  by induction on the level of recursion in the computation of N[E].
       basis: if E is a number, a primitive, or a lambda expression then N[E]=E.
       induction: we assume that the Theorem holds for recursive calls to N.
       Then the Theorem holds for the remaining syntactic cases of E by the
       monotonicity of ->.


We note in passing that N[E] is not necessarily a normal form.  Lambda
expressions, in particular, are not reduced by N, since otherwise the
evaluation of certain meaningful expressions (e.g. the paradoxical combinator
Y) would not terminate.


2.6:  Functional Domain of N

In this section it is shown that the entire domain of N constitutes a
functional domain satisfying the intuitive criteria of [1.1].  We interpret
the semantic equivalence relation, ~, on the domain of N as follows:

$$\text{For } X,Y \text{ in } D_N, \quad X{\sim}Y \text{ if and only if} \qquad\qquad [2.10]$$
$$\text{for every } Z \text{ in } D_N \text{ and number } n,$$
$$(Z\ X)=n \quad <=> \quad (Z\ Y)=n$$

where $D_N$ is the domain of N.  We now justify this interpretation of ~ on N
thru


Thm 2.11:  The domain of N is a functional domain, obeying the criteria of
           [1.1], under the above interpretation of ~.

    proof: The equivalence relation ~ defined in [2.10] must be shown to obey

the three clauses of [1.1] over the domain $D_N$ of N.  We treat the clauses
individually:

1) For numeric constants X and Y, we must show that X~Y <=> X=Y.

   <=: direct, by the equivalence of identical expressions.

   =>: Assume X~Y.  Then by beta-reduction,

$$((LAMBDA(a)a) X)=X$$

and

$$((LAMBDA(a)a) Y)=Y$$

and thus, by [2.10], X=Y since they are numeric.  By [2.7] there exists a
Z such that X and Y are each reducible to Z; since X and Y are not
reducible, Y, Y, and Z must be identical.

3) To show: X~Y <=> for all Z in $D_N$,
                                (Z X)~(Z Y) or neither defined.

   =>: Assume false.  Then for some X~Y there exists a $Z_1$ such that

$$(Z_1 X)\dagger(Z_1 Y)$$

where $\dagger$ is the negative of ~.  This implies, by [2.10], that there exists
a $Z_2$ such that

$$(Z_2 (Z_1 X))=n$$

for some numeric constant n but <u>not</u>

$$(Z_2 (Z_1 X))=n$$

(we are assuming here one of two completely symmetric cases with no loss
of generality - the other case follows by interchanging the symbols X and
Y).  Defining $Z_3$ by the lambda expression

$$Z_3 \underline{=} (LAMBDA(a)(Z_2 (Z_1 a)))$$

we note that

$$(Z_3 X)=n \quad but \quad (Z_3 Y)\neq n$$

hence by [2.10] X$\dagger$Y.

<=: Assume that for all Z in $D_N$, (Z X)~(Z Y). Then (Z X)=n (for numeric constant n) if and only if (Z Y)=n by the argument of part (1). Hence by [2.10] X~Y.

2) It must be shown that X~Y if and only if for all Z in $D_N$, (X Z)~(Y Z). From part (2) of this proof, X~Y <=> for all Z:

$$((LAMBDA(a)(A\ Z))\ X)~((LAMBDA(a)(a\ Z))\ Y)$$

hence, by beta-reduction,

$$(X\ Z)~(Y\ Z)$$

The significance of Theorem 2.11 is that _every_ element of the domain of N corresponds to some element of the abstract semantic domain: every element of $D_N$ is intuitively functional. Thus N (and the lambda calculus on which it is based) is a language of "pure" functions. We shall find in the next chapter that this pleasant property costs us something, however, in terms of expressive power.


2.7:  Summary

The material in this chapter is largely introductory. The three interpreters presented are abstracted from conventional implementations, and their scrutiny serves to relate common implementation issues to the expressibility of functions. The major findings were:

1) Each language is arithmetically complete, in the sense that every computable function defined on the natural numbers is expressible.

2) The FUNARG problem leads to the inexpressibility in S of functions whose domain contains integers and whose range contains infinitely many functions.

3) Applicative order evaluation renders inexpressible in T every function whose domain includes *, the undefined computation. An example of such a function is the constant function (LAMBDA(X)3) of one argument.

4) The interpreter N, based on the normal order evaluation of expressions by substitution, suffers from neither of these deficiencies. We can construct a functional domain F such that every expression X in the domain of the language N corresponds to an element of F; thus N is a "pure" language in the sense that every expression corresponds to a function or a number. This is not true, for example, in S, where lambda expressions containing free variables can compute different functions in varying contexts.

We are left with N, an interpreter whose behavior is intended to model the lambda calculus; the remainder of this report, roughly speaking, deals with a particular weakness common to N and the lambda calculus.

## Chapter 3:
## Motivation for a Multi-valued Semantics

Central to this chapter is the argument that the N model, and hence the lambda calculus, is functionally incomplete because of the inexpressibility in N of a class of computable functions on N's domain. The inadequacies of N leading to this weakness are explored, and two new model languages are presented, each curing the problem in a different manner. The first model, which has provision for encoding representations of functions as integers, is found to be unsatisfactory for both practical and semantic reasons. The alternative solution proposed in this chapter involves mechanism for the representation of semantic elements with multiple values; this mechanism, called EITHER, is the principal focus of the remainder of the Thesis.

### 3.1:  Necessity of non-functions: WHICHFF

Consider the family of partial functions, $\{FF_i\}$ for i ranging over N, which satisfy the following conditions: for each natural number $\underline{i}$,

$$FF_i[x] = i, \ i=x \hspace{4cm} [3.1]$$
$$\text{divergent}, \ i{\neq}x$$

Thus each $FF_i$ has a single element in its domain: the number $\underline{i}$. For any other argument the value of $FF_i[x]$ is undefined. The $\{FF_i\}$ are clearly partial functions in the intuitive sense of Defn [1.1], and are computable in each of the model languages considered here. Furthermore, they are semantically distinct: for no numbers $i{\neq}j$ does $FF_i{\tilde{}}FF_j$. There is then nothing intuitively objectionable about a function which maps each $FF_i$ to its corresponding $\underline{i}$. Consider such a function WHICHFF which, for each natural number $\underline{i}$, has the property that:

$$WHICHFF[FF_i] = i \hspace{4cm} [3.2]$$

Intuitively WHICHFF is a function from $\{FF_i\}$ onto N; furthermore it is demonstrably computable using "dovetailing" or multiprocessing techniques. Note in particular that the following definition of WHICHFF satisfies the condition of [3.2]:

WHICHFF[f] = i such that f[i]=i,                                    [3.3]

if such a number i exists;

else undefined

We may view the dovetailed evaluation of WHICHFF[f] as the computation of f[0] for one second, the computations of f[0] and f[1] each for two seconds, and similarly until any one of the computations f[i] terminates normally; the value of this f[i] would then be taken as the value of WHICHFF[f]. However, WHICHFF is not expressible in N; this is a result of

Thm 3.4: Let L be an arithmetically complete applicative language and let $D_L$ be the domain of L. Then no function WHICHFF having the properties of [3.3] is functional over $D_L$.

proof by reduction to the halting problem. Assume that $D_L$ contains a function WHICHFF having the property given in [3.3]. Then for any function $f$ in $D_L$ and any number $i$, L[(WHICHFF $f$)]$^-i$ if $f^-FF_i$. Now consider the union of the functions $FF_1$ and $FF_2$, given by:

$$FF_{12}[x] = 1, \quad L[x]=1;$$
$$2, \quad L[x]=2;$$                                                [3.5]
$$\text{divergent otherwise}$$

$FF_{12}$ is clearly a computable first order function, hence it is expressible in L by the arithmetic completeness of L. Now L[(WHICHFF $FF_{12}$)] can have as its value at most one of {1,2}; thus either L[(WHICHFF $FF_{12}$)]$\neq$1 or L[(WHICHFF $FF_{12}$)]$\neq$2. Assume, with no loss of generality, the former. Then define the second order function $g$ as follows:

$$g[f] = \text{the function } g_f, \text{ where}$$
$$g_f[i] = \quad 1, \quad i=1;$$
$$2, \quad i=2 \text{ and } f[0] \text{ defined};$$
$$\text{divergent otherwise. For every computable}$$

first order function $f$, $g_f$ (or equivalently g[f]) is evidently computable. Moreover, if f[0] is undefined then $g_f$ is identical to the function $FF_1$; otherwise $g_f$ is identical to the function $FF_{12}$. We use the ability of WHICHFF to distinguish between $FF_1$ and $FF_{12}$ to determine whether f[0] is defined, by means of the function $h$ given by

$$h[f] = WHICHFF[g[f]]$$

We note finally that for any function $\underline{f}$

$$f[0] \text{ convergent} \Rightarrow g[f] \sim FF_{12} \Rightarrow h[f] \neq 1;$$

and

$$f[0] \text{ divergent} \Rightarrow g[f] \sim FF_1 \Rightarrow h[f] = 1$$

Hence $h[f]=1$ if and only if $f[0]$ is divergent. The divergence of $f[0]$ is decidable, as one of the computations $h[f]$ and $f[0]$ must converge; thus the function h provides a solution to the "halting problem" for first order functions, and is a well known noncomputable function. Since h is clearly computable in terms of WHICHFF, we conclude that WHICHFF is not a computable function over any domain including the first order functions.

Since it was shown in the last chapter that _every_ function expressible in N is functional over all of the domain of N, it follows that WHICHFF is not expressible in N. This inexpressibility relates intuitively to two aspects of the implementation of the N interpreter:

1) The interpreter does not admit multiprocessing. If, in the evaluation of expression A, N embarks on the evaluation of a subexpression $\underline{a}$ of $\underline{A}$ whose N value is not defined, then the N value of $\underline{A}$ is not defined.

2) The only mechanism in N by which a function $\underline{f}$ can recover information about its functional argument $\underline{g}$ is the application of $\underline{g}$. There is no means by which $\underline{f}$ can discover the algorithm (or program) by which $\underline{g}$ computes values, even though the internal representation of $\underline{g}$ necessarily includes this information. Hence if $\underline{f}$ is to make any use of $\underline{g}$, then $\underline{g}$ must be applied to some argument; By the constraint (1) above, the nontermination of this application results in the nontermination of the application of $\underline{f}$.

The correction of either of these deficiencies is straightforward in an implementational sense -- many extant languages boast provisions for multiprocessing and/or access to representations of functions. However, neither "correction" is easily reconciled with the semantics of an applicative language. The second limitation of N seems a natural consequence of our

distinction between the notions of a _function f_ and any of the _algorithms_ for computing _f_ from its arguments; a language which provides mechanism for distinguishing between algorithms for computing a particular function _f_ would certainly have non-functional elements in its domain. The semantic ramifications of a cure to the first problem, however, are more subtle and will be explored in detail.

The following sections present two alternative extensions to N, each corresponding to a "fix" of one of the above limitations. The function WHICHFF is expressible in each.

## 3.2: Coding primitives: The C model

We noted that a limitation of N, justifiable by our intuitive respect for the semantics of functions, is that no information can be recovered about an N function without the application of that function. In particular, N provides no means for recovery of information about the representation of a function as an N expression. We have thus avoided the "Turing machine tar pit" -- the argument that any language as powerful as a Universal Turing Machine has exactly the same set of expressible functions.

The C model presented here has, in addition to the primitives and structure of N, primitives for the translation of the representation of language elements to and from a tractable form. Making the fundamental assumption that any function defined on a domain F is computable if and only if it is computable from the representations of elements of F, we must conclude that a Universal Turing Machine (or its equivalent) operating on the representations of arguments to the computable function _f_ can compute representations of the values of _f_. This is the substance of our claim of functional completeness of the language C.

The interpreter for C is identical to the interpreter for N except for the addition of the primitive operators CODE and DECODE. CODE maps representations of the domain of C into the natural numbers:

$$\text{CODE: } D_C \to N$$

and may be viewed as a Goedelization of the character string representing its
argument.  The claim we make for CODE is that if (CODE X) and (CODE Y) have
the same (numeric) value then X and Y are semantically equivalent;  they are
in fact represented in an identical manner.  We cannot, of course, claim that
in general X~Y implies (CODE X)=(CODE Y), as there are many representations of
each semantic element and the semantic equivalence of the representations is
effectively undecidable.  The operator DECODE is the inverse of CODE:  given
the Goedel number of the representation of an element, it returns the element.
We thus claim that each expression X is semantically equivalent to (DECODE
(CODE X)).

Our claim for the functional completeness of C is formalized, to the extent
possible, in

Thm 3.6:  Let F be a functional domain of C, and let

$$g: F \rightarrow F$$

be a computable function on F.  Then g is expressible in C, i.e., there
is an expression G in the domain of C such that for all x,y in F, g[x]=y
implies that (G X)~Y.

proof: Since g is computable then so is h defined by:

$$h \doteq (LAMBDA(Y)(CODE\ (g\ (DECODE\ Y))))$$

as it is simply the composition of computable functions.  Furthermore,
since h is a function from N to N, it is expressible in C by the
arithmetic completeness of C; let H be the representation in C of h.
Then the function g is expressible in C by:

$$G \doteq (LAMBDA(X)(DECODE\ (H\ (CODE\ X))))$$

It must be recognised that CODE is not functional: it radically disobeys the
intuitive requirements of Defn 1.1.  We note, for example, that CODE might
return different values for the arguments (LAMBDA(X)X) and (LAMBDA(Y)Y) as
they have different representations, violating our requirement that
semantically equivalent arguments produce semantically equivalent results.

WHICHFF example of the preceding section.  The representation of WHICHFF in C
involves writing an interpretor, operating on the CODEd representations of C
expressions, which simulates the required "dovetailing" by computing 1 step of
(g 1), 2 steps of (g 2), 2 steps of (g 1), etc.  Presentation of actual code
for WHICHFF on C would be, at best, a messy task; it is hoped therefore that
the reader will accept the expressibility of WHICHFF in C on the basis of
Theorem 3.6 and this informal discussion.


### 3.2.1:  The Turing-machine Tar Pit

The introduction of the specter of coding requires further reflection.  We
have made the enticing observation that, with the introduction of a simple
mechanism allowing the representations of functions to be accessible as data,
every computable function becomes expressible.  We have noted corollary
disadvantages -- (i) the semantic confusion resulting from the nonfunctional
character of CODE, and (ii) the practical absurdity of having to include the
code for **interpreters** in the definitions of certain functions.

However, the inclusion of coding primitives in an applicative language may be
objected to on more fundamental grounds than the above.  The stated semantic
goal of an applicative language is the representation of functions.  Thus such
a language provides a set of rules and conventions for associating expressions
with abstract functions; moreover, the power and consistency of the language
stem largely from the applicability  of these rules and conventions to every
expression in the language.  In the lambda calculus, for example, we are
assured that expressions which are interconvertible via the alpha and beta
axioms are equivalent.  The cost of this assurance is a corresponding
constraint on the computations which we might perform: the alpha axiom
positively prohibits us from writing a function which distinguishes
(LAMBDA(X)X) from (LAMBDA(Y)Y).  We accept this constraint because the
structure which it imposes is useful to us; we recognize that we cannot be
assured of a relation and simultaneously be allowed to violate it at will.

Coding primitives may be viewed as a mechanism for violating the structure
imposed by an applicative language.  None of the lambda calculus axioms, for
example, are valid in the presence of coding, since "functions" can be written

which distinguish between interconvertable expressions.  The rules and
conventions for representing functions are, in effect, abandoned.  The
programmer is thus freed from the structural constraints of the language, but
finds himself in a semantic anarchy -- while he may write any function he
pleases, he may make no assumptions about the structure or representation of
its arguments.


### 3.2.2:  Functionality of DECODE

We may convincingly defend the contention that CODE is not a function by
demonstrating that it returns semantically distinct integers, say, for the
equivalent arguments (LAMBDA(X)X) and (LAMBDA(Y)Y).  This demonstration does
not apply, however, to the inverse of CODE;  there is nothing inherently
nonfunctional in the fact that DECODE returns semantically equivalent
expressions (LAMBDA(X)X) and (LAMBDA(Y)Y) when given semantically distinct
integers as arguments.  It is the purpose of this section to demonstrate that
functions with the property of DECODE (i.e. mapping a subset of the natural
numbers onto the entire domain of discourse) are expressible in N and the
lambda calculus.


### 3.2.2.1:  LAMBDA-free AEs

It is convenient for certain purposes to use the techniques developed
primarily by Curry[12] of the calculus of __combinators__ for the reduction of
applicative expressions to equivalent expressions whose use of lambda
expressions is highly restricted.  For our purposes we shall consider the
combinators listed below (along with their respective definitions):

$I \equiv$ (LAMBDA(X)X)

$K \equiv$ (LAMBDA(X)(LAMBDA(Y)X))

$W \equiv$ (LAMBDA(X)(LAMBDA(Y)(X Y)))

$S \equiv$ (LAMBDA(X)(LAMBDA(Y)(LAMBDA(Z)((X Z)(Y Z)))))

$G_1 \equiv$ (LAMBDA(G)(G G))

$G_2 \equiv$ (LAMBDA(G)(LAMBDA(Y)(Y G)))

$G_3 \equiv$ (LAMBDA(Y)(LAMBDA(X)((Y X) X)))

$G_4 \equiv (LAMBDA(G)(LAMBDA(D)(LAMBDA(X)(G (D X)))))$

We show in this section that every applicative expression using no lambda expressions other than the above combinators; we begin with

Lemma 3.7:  Let R be a LAMBDA free AE in the single argument applicative language L, and let R contain occurrences of the variable x.  Then R is equivalent (by alpha and beta axioms) to a LAMBDA free AE of the form (R′ x) where R′ contains no occurrences of the variable x.

  proof is by structural induction on R.

    basis: R is atomic (in particular, R is not a combination).  If r is the variable x, then r′ is (I x)=x (by axiom beta).  If r is not the variable x, then r contains no free occurrences of x and r′ is ((K r) x) = ((LAMBDA(X)r) x) = r.

    induction: R is a combination of the form $(R_1 R_2)$.  By inductive hypothesis, $R=((R_1′ x)(R_2′ x))$ for some AEs $R_1′$ and $R_2′$ not involving the variable x; then $R′=(((S R_1) R_2) x) = ((LAMBDA(Y)(LAMBDA(X) ((R_1 X)(Y X)))))) = ((R_1 x)(R_2 x))$.

The principal result of this section is the following adaptation from Curry's Synthetic Theory of Combinators:

Thm 3.8:  Let A be an AE in a single-argument applicative language L whose semantic equivalence obeys axioms alpha and beta.  Then A is equivalent to a LAMBDA-free expression A* containing only the combinators I, K, W, S, $G_1$, $G_2$, $G_3$, $G_4$, and the primitives and constants of L.

  proof: We show that, given any such A which is not LAMBDA-free, we can construct an equivalent A′ containing fewer LAMBDAs.  Let a be an innermost LAMBDA expression occurring as a subexpression of A.  We then construct A′ by replacing a as follows:

    Case 1: a is of the form (LAMBDA(x)x) for some variable x; we replace a by I (equivalent by axiom alpha).

Case 2: <u>a</u> is of the form (LAMBDA(x)y) where x and y are different variables; we replace <u>a</u> by (K y).

Case 3: <u>a</u> is of the form (LAMBDA(x)(b x)) where x is a variable and b is an AE: replace <u>a</u> by (W b)=(LAMBDA(Y)(b Y))

Case 4: <u>a</u> is of the form (LAMBDA(x)(c d)): By Lemma 3.7, the body (c d) is equivalent to an AE (r´ x) where the variable <u>x</u> does not appear in r´. Then <u>a</u>=(LAMBDA(x)(r´ x)) which is reducible according to case 3.

Since each expression A which is not LAMBDA free is thus equivalent to an expression A´ containing fewer LAMBDAs, a finite number of such reductions will reduce each such A to a LAMBDA free A*. This completes the proof.

It is a relatively simple exercise to show in addition that each of the combinators I, W, $G_1$, $G_2$, $G_3$, $G_4$ is in turn equivalent to an expression in K and S, allowing us to simplify Theorem 3.8 by eliminating all but 2 of the combinators. This is unnecessary for our purposes, however, so long as the number of combinators required is finite. An important observation to be made at this point is that the construction of A* detailed in Theorem 3.8 is <u>effective</u>; thus we could program a computer to convert AEs to LAMBDA free form.

## 3.2.2.2: An Enumeration of $D_N$

In this section it is demonstrated that the domain of every applicative language with the power of the N model contains functions which enumerate the domain of that language, ie, each such language L with domain $D_L$ contains a function

$$f: N \to D_L$$

such that for every finite expression <u>x</u> in $D_L$ there is a number <u>n</u> which satisfies (f n)=x. We procede by Goedelizing the LAMBDA free expressions of $D_L$.

Let pair be a number pairing function such that, for each i and j in N, the
value of pair[i,j] is a unique number $P_{ij}$, and let left and right be functions
recovering the components of a pair; ie, for every i and j, left[pair[i,j]]=i
and right[pair[i,j]]=j. There are many well known such pairing functions;
since they are all first order computable functions, we may assume that they
are expressible in each of our model languages.

Let us now suppose that we label the (finitely many) primitives of the
language L as $p_1$, $p_2$, ... $p_n$. Note that we include the combinators K, I, $G_1$,
etc. in this list so that we can enumerate LAMBDA free expressions only. We
now specify the coding details: for each LAMBDA free expression x, we define
the Goedelization g[x] as follows:

    g[x] =
        if x is a number then pair[0;x];
        if x is a primitive $p_j$ then pair[1;j];
        if x is a combination (a b) then pair[g[a];g[b]];

The function g is computable from the representation of x, but we cannot in
general claim that it is computable from the functional properties of x. The
function g is, in fact, a satisfactory choice for the CODE function of the C
model, assuming (as we may) that we are content to deal with LAMBDA free
expressions of C. If such a function g could be shown to be computable in,
for example, the N model, we would have a direct a priori demonstration that
the languages are expressively equivalent. We must, however, be content with
the expressiblity of a semantic inverse of g: the function enu defined such
that, for every LAMBDA-free expression x, enu[g[x]]=x. This apparent
asymmetry can be explained by the observation that g is not a function, in the
sense of Defn [1.1] which prohibits the mapping of semantically equivalent
expressions into differing numbers. The fact that enu may map different
numbers into semantically equivalent values is consistent with its
functionality. We label the expressibility of enu as

Thm 3.9:  Let L be an extension of N with primitives 11, 12, ..., 1n
         (including combinators K and S). Then there is a function enu:N->$D_L$ such
         that, for every LAMBDA free expression x in $D_L$, there is a number i such
         that enu[i]~x.

<u>proof</u> is a straightforward programming job.  Such a function for the
language N would take the form:

```
(LAMBDA(N)((GREATER (LEFT N) 1)
          ((ENU (LEFT (RIGHT N)))
            (ENU (RIGHT (RIGHT N))) )
            ((GREATER (LEFT N) 0)
                ((GREATER (RIGHT N) n-1) ln

                ...

                   11)...))
            (RIGHT N) )))))
```

where li is the <u>i</u>th primitive of N, and LEFT and RIGHT are the N
expressions corresponding to the <u>left</u> and <u>right</u> functions above.

## 3.3:  E model: Multiprocessing primitives

An extension to the N interpreter which is somewhat more palatable than the
use of coding primitives is the addition of mechanism for multiprocessing: the
quasi-simultaneous evaluation of several expressions.  We consider here the E
model, which is the N model of Chapter 2, augmented by the primitive operator
EITHER whose interpretation is as follows:

For every choice of expressions <u>a</u> and <u>b</u>,                              [3.10]

> E[(EITHER <u>a</u> <u>b</u>)] =
>
> > if E[<u>a</u>] is defined but E[<u>b</u>] is not, then E[<u>a</u>];
> > if E[<u>b</u>] is defined but E[<u>a</u>] is not, then E[<u>b</u>];
> > if E[<u>a</u>] and E[<u>b</u>] are both defined then one of these values;
> > else undefined.

Note that  we do not specify which of the arguments is returned if both have
defined values;  we may consider that this selection is made by some
nondeterministic process over which we have no control.  EITHER is evidently
computable by dovetailing techniques, eg by evaluation of E[<u>a</u>] and E[<u>b</u>] each
for 1 step, then each for 2 steps, and so on until one evaluation or the other
returns a value.  EITHER is not, however, functional: in the case where <u>a</u> and

b each have defined values (and their values differ), then the value of
(EITHER a b) is dependent on the representation of a and b and on details of
scheduling of the dovetailed computation.

The power of the either primitive is demonstrated by the expressibility of
WHICHFF in E as follows:

$$\text{WHICHFF}[f] = g_1[f;0]$$

where $g_1[h;n] = \text{either}[h[n];g_1[h;n+1]]$

Note that for $i>j$, $g_1[FF_j;i]$ is undefined and hence for $i \leq j$ $g_1[FF_j;i]=j$. Thus
for every number j, $E[(\text{WHICHFF } FF_j)]=j$.

The presentation of the EITHER primitive in this section is informal, based
largely on its intuitive relation to the implementation mechanism of
multiprocessing. The formalization of this mechanism is a principal topic of
the remaining chapters. The remainder of the present chapter explores the
impact of EITHER on the semantics of an applicative language.


3.4:  The Intuitive Paradox

The reader has doubtless noticed that fundamental questions raised in the
first section of this chapter demand a more precise characterization of the
hitherto vauge notion of functional completeness. Specifically, Theorem 3.4
shows that WHICHFF is not functional over the entirety of any functional
domain which includes all first order functions. Thus the basic intuitive
requirements of [1.1] are inconsistent with the existence of a functional
domain F which is arithmetically complete and includes every computable
function f:F->F. Two alternatives facing us are the following:

1) We can deny that WHICHFF is a computable function. Indeed, Theorem 3.4
   may be interpreted as a statement that no computable function defined on
   first order functions has the properties of WHICHFF given in [3.2]. Our
   intuitive claim that WHICHFF is a computable function is based on the
   incomplete specification of its behavior over the entire functional
   domain: [3.2] merely defines it over the restricted domain of $\{FF_i\}$.

2) We can revise the notion of a functional domain F such that, for every function $f$ in F there is a <u>domain of specification</u> over which the behavior of $f$ is defined. The functional criteria of [1.1] are then required to apply only when the arguments of $f$ are drawn from its domain of specification, $S_f$.

3) We can postulate new elements of the functional domain F corresponding to the values returned by otherwise nonfunctional procedures.

We reject the first choice on the grounds that it restricts our consideration to those functions expressible in the lambda calculus, giving us no way of distinguishing between N and the intuitively superior E. The second choice is rejected after brief consideration (in a following section) partly because of the technical complications it entails, but primarily because it denies the semantic validity of the interesting class of multivalued expressions. The third choice seems the most promising from the point of view of rigorous analysis, but requires a substantial intuitive leap whose usefulness must be carefully scrutinized. This project is approached in subsequent sections.

## 3.5: Multi-valued Semantic Elements

The domain $D_N$ of language N was shown, in Chapter 2, to have the property that every element $x$ of $D_N$ corresponds to exactly one element of a functional domain; thus each expression $x$ in $D_N$ has, intuitively, exactly one semantic value or meaning.[1] In this chapter it was shown that this graceful property of $D_N$ is inconsistent with the expressiblity of the function WHICHFF, a demonstrably computable and intuitively well behaved function over a particular subset of $D_N$. Our implementation of WHICHFF, while functional over this restricted domain S, behaves poorly when given arguments from $D_N$ which are not in S; furthermore, this annoying defect is characteristic of <u>every</u> implementation of WHICHFF in a language sufficiently powerful as to be arithmetically complete. The problem is evident when WHICHFF is applied to the function $FF_{12}$: either of the values 2 or 3 is consistent with the

---

[1] It must be recalled that we have postulated a semantic element, *, corresponding to the "meaningless" or nonterminating computation; hence a possible semantic value for $x$ is *.

definition of WHICHFF [3.3], and there is no implementation of WHICHFF which consistently returns a single value, eg 2, when applied to every $\underline{x}$ in $D_N$ semantically equivalent to $FF_{12}$. Thus the evaluation of (WHICHFF $FF_{12}$) leads to exactly the same underdetermined result as the evaluation of (EITHER 1 2): the E values of each expression might be 1 or 2, depending on circumstances which are irrelevent to the semantics of each expression.

### 3.5.1: Domains of Specification

One means of avoiding such apparently nondeterministic computations is to exclude them from our semantic model, ie, to deny that (EITHER 1 2) has any semantic value. Under this restriction, we must carefully exclude from our consideration any expression having multiple E values, either by avoiding the use of EITHER and reverting to the well behaved domain $D_N$, or by assuring ourselves, at each application of EITHER, that the result is single valued. We may note, pursuant to the latter program, that for all expressions $\underline{a}$ and $\underline{b}$, E[(EITHER $\underline{a}$ $\underline{b}$)] is single valued if

1) $\underline{a}$ is single valued and $\underline{b}$ is meaningless; or
2) $\underline{b}$ is single valued and $\underline{a}$ is meaningless; or
3) $\underline{a}$ and $\underline{b}$ are both meaningless; or
4) $\underline{a}$ and $\underline{b}$ are each single valued and their values are semantically equivalent.

So long as the arguments to EITHER satisfy the above criteria, EITHER is intuitively functional. For each function $\underline{f}$ whose definition involves EITHER, we may then carefully define a domain of specification $S_f$ such that for arguments $\underline{x}$ from $S_f$, E[(f x)] is single valued. We may, for example, show that our definition of WHICHFF in terms of EITHER is functional over a domain of specification including the functions $\{FF_i\}$.

This means of avoiding the semantic difficulties of EITHER may raise certain aesthetic objections. First, it places on us the considerable burden of having to construct domains of specification for each of a large class of functions, and the necessity of showing that each such function is well behaved over its particular domain of specification. Second, it rules out

consideration of algorithms for well behaved functions which have
multiply-valued subexpressions.  Consider, for an example of the latter
limitation, the function $\underline{f}$ defined so that

> f[n] = 5, n=1
>        5, n=2
>        else undefined.

Now, since $\underline{f}$[1]=5 and $\underline{f}$[2]=5, it is intuitively reasonable to claim that
$\underline{f}$[either[1;2]]=5; yet we cannot make such a claim unless we are willing to
assign some semantic value to either[1;2].


## 3.5.2:  EITHER and the Lambda Calculus

There is an essential incongruence between EITHER and the axiomatic basis of
the Lambda Calculus which precludes the incorporation of the former as a
primitive with an associated <u>delta</u> rule.[1]  Recalling that these axioms define
an equivalence relation, =, on the domain of the language, incorporation of
EITHER results in the equivalences:

> (EITHER 1 2)=1
> (EITHER 1 2)=2

and hence

> 1=2

from which it follows, by the famous logic of Russel, that "I am the Pope".
Clearly the relation between (EITHER 1 2) and 1 is not equivalence, but rather
some irreversible reducibility property.  Any evaluator which can yield 1 as
the value of (EITHER 1 2) cannot be claimed to preserve semantic equivalence;
it merely <u>reduces</u> that expression to one of its several values and discards,
in the process, information about the other values.  This is the underlying
reason why N (and the Lambda Calculus) are incapable of expressing WHICHFF,
and is basic to the proof of Theorem 3.4.

---

[1] Such a delta axiom is formally ruled out by the requirement that the
arguments to primitives be in <u>reduced</u> form, thus restricting applications of
EITHER to cases where both arguments have meaningful E values.

## 3.6:  The Power Set Domain

The natural extension of a functional domain F of single-valued elements to a
domain F* of multiply-valued elements involves the interpretation of F* as the
power set, or set of subsets, of F.  Thus the elements 2 and 3 of F correspond
to the unit subsets {2} and {3}, respectively, in F*, while the semantic
element of F* corresponding to the value of (EITHER 2 3) is the subset {2,3}
of F containing both 2 and 3.  The meaningless element * corresponds to the
empty subset $\emptyset$ of F, having no value.  Other useful relationships which we
would like to see in F* include the following:

1) If $a \tilde{\ } b$ in F then $\{a,b\} \tilde{\ } a \tilde{\ } b$ in F*.

2) (EITHER ($f$ $a$)($f$ $b$))$\tilde{\ }$($f$ (EITHER $a$ $b$)), or equivalently, the elements
   {f[a],f[b]} and f[{a,b}] in F* are the same.

3) The natural interpretation of either on functions leads to the semantic
   equivalence (EITHER f g)$\tilde{\ }$(LAMBDA(X)(EITHER (f X)(g X))).  This allows us
   to propose, in symmetry with (2), that:

4) ((EITHER $f$ $g$) $a$) $\tilde{\ }$ (EITHER ($f$ $a$) ($g$ $a$)).

5) (EITHER $a$ *)$\tilde{\ }$$a$, where * is the element corresponding to the undefined
   computation.

6) If $a$ corresponds to $\{a_1,...,a_j\}$ in F* and $b$ corresponds to $\{b_1,...,b_k\}$,
   then (EITHER $a$ $b$) corresponds to $\{a_1,...,a_j, b_1,...,b_k\}$ in F*.  In
   general, EITHER of multivalued elements corresponds to the union of the
   respective elements of F*.


## 3.7:  Interpretation of F*

The semantic model being developed in this chapter demands a certain amount of
intuitive realignment on the part of the reader.  The attractive feature of F*
as a semantic domain is that it allows the preservation of a notion of
semantic equivalence, without cost in terms of expressibility of certain
functions.  Its major disadvantage, at least from an intuitive standpoint, is
that it requires that we postulate certain abstract semantic elements which

are intangible in practice -- if the expression $\underline{x}$ has multiple values, say 2
and 3, then we have no way of discerning from the value "3" typed by our E
interpreter that "2" is also a value of $\underline{x}$. We could, of course, build an
interpreter which would <u>enumerate</u> the values of $\underline{x}$ by dovetailing computations
at each EITHER juncture. However, as $\underline{x}$ might have infinitely many values,
this process may never terminate; worse yet, even for an x with finitely many
values we cannot tell, in general, when all of the values have been typed.

There are, however, situations where this ambiguity is unimportant. We may
know, for example, that $\underline{x}$ is single valued, in spite of the dual values of a
subexpression $\underline{y}$ of $\underline{x}$. Alternatively, we may recognise that $\underline{x}$ has many values,
but be willing to settle for any one of them.


## 3.8: Computable elements of F*

If we have a procedure for identifying the computable elements of a single
valued domain F, we can characterize the computable elements of the power set
domain F* as those elements of F* which are effectively enumerable sets of
computable elements of F. Given an expression X we can enumerate the
components of the F* element representing X; one means of doing so is provided
in Chapter 6. Furthermore, given an expression G for a function which
enumerates a set S of elements of F, we can construct an expression whose
representative F* element is S; take for example the expression

            ((Y (LAMBDA(H)(LAMBDA(X)(EITHER (G X)(H (PLUS 1 X)))))) 0)

where Y is the fixed point operator (LAMBDA(F)((LAMBDA(G)(F (G
G)))(LAMBDA(G)(F (G G))))). This expression reduces to an expression of the
form

        (EITHER (G 0)
            (EITHER (G 1)
                (EITHER (G 2)
                    (EITHER (G 3) --- ))))

and its corresponding element of F* is exactly the range of G.

We may use as our function G in the above expression an enumerator ENU of the entire domain F, constructed by the techniques of section 3.2.1.2; this expression, TOP, corresponds to the semantic element of F* which is the set F itself.

## 3.9: Summary

This chapter raises the question of the expressibility of a particular function, WHICHFF. This function is inexpressible in the lambda calculus, and intuitively it requires a mechanism for multiprocessing for its implementation in spite of its applicative -- hence time independent -- nature. Two alternative extensions of the N interpreter are proposed, each of which renders WHICHFF expressible:

1) Primitives can be added to N which allow coding and decoding of arbitrary expressions into and from numbers. This mechanism allows programs to access the representation of functions, and it is argued that such a CODE/DECODE facility extends any arithmetically complete language to functional completeness. Yet the use of this mechanism is awkward: the specific implementation of WHICHFF, for example, requires coding an interpreter which simulates the necessary multiprocessing. Moreover the semantic ramifications of CODE are drastic, involving abandonment of much of the applicative structure of any language in which it is embedded.

2) A primitive, EITHER, can be added to N to implement multiprocessing. EITHER renders WHICHFF easily expressible, and it may be justified semantically in an applicative language.

In connection with (1), it is noted that although the new primitive CODE is radically nonfunctional, the inverse operation of DECODE (which maps codings into the functions which they represent) is acceptable as an element of our functional domain. A combinatory proof shows that such decoding functions are, in fact, expressible in the unmodified N language; hence we can write in the lambda calculi functions which enumerate the entire semantic domain of these calculi.

The introduction of EITHER or equivalent mechanism requires that we modify the structure of the semantic domain and its relation to expressions of a language. In particular, it seems most natural to associate with each expression a _set_ of abstract values, rather than a unique single value. We thus move from the domain F of single values to the domain $F^*$ whose elements are enumerable subsets of the elements of F; we term $F^*$ the power set domain.

The presentation of EITHER in this chapter is informal and relies heavily on implementational notions such as multiprocessing. The following chapters formalize the mechanism in terms of systems of conversion rules, based on the lambda calculus; this process both justifies and refines the rough implementation model sketched here.

## Chapter 4:

## Theories of EITHER-conversion

While the implementation and semantic considerations of the previous chapter
provide a strong intuitive basis for the interpretation of EITHER, the further
development of this new mechanism requires something more concrete.
Specifically, the incorporation of EITHER into a language E involves syntactic
manipulations of expressions in E, and hence necessitates a formalism which
distinguishes those syntactic manipulations which are semantically valid from
those which are not. The relationships developed in the last chapter are
analogous to the convention that "(PLUS 2 3)" represents the sum of 2 and 3,
without a corresponding mechanism for associating this expression with the
expression "5".

This chapter begins the project of developing formalisms, i.e. conversion
axioms, for the syntactic manipulation of expressions involving EITHER.
Several theories (i.e., systems of axioms) are presented in this and
subsequent chapters; each is based on the beta-delta[1] calculus, with
additional axioms for manipulation of the new EITHER construct. The
distinction between these theories stems from an issue of evaluation order,
discussed in a following section, and reflects alternative interpretations of
certain expressions involving EITHER.

A principal difference between the axiom systems presented here and those of
the lambda calculus is the introduction of a new asymmetry, in the form of an
ordering relation $\geqslant$, between expressions of E. We have seen in previous
sections that it is futile to require that E interpretation preserve an
equivalence relation; such a requirement was shown to lead to an
inconsistency in any language capable of expressing WHICHFF, since (WHICHFF
$FF_{12}$)~1 and (WHICHFF $FF_{12}$)~2 together imply that 1~2. The asymmetry of $\geqslant$,
however, allows the relations (WHICHFF $FF_{12}$)$\geqslant$1 and (WHICHFF $FF_{12}$)$\geqslant$2 to hold
without compromising the semantic relation between 1 and 2. We view the
relation $\geqslant$ as designating EITHER-reducibility, and may interpret x$\geqslant$y
informally to mean that the values of y are among the possible values of x.

---

[1] No attempt is made to incorporate eta conversion into the systems presented
here, although we expect that no new difficulties would arise in doing so.

We shall use x≡y to mean that both x>y and y>x.

It is important to distinguish between the relation > and the "reducible to"
relation, ->, of the lambda calculus. If the expression X is reducible to the
expression Y by means of conventional lambda calculus axioms, then it will
follow that X>Y and Y>X; the reverse, however, is not true. The semantic
interpretation of X>Y is that every value of Y is also a value of X; i.e., the
element of F* corresponding to Y is a subset of the element corresponding to
X.

## 4.1: Preliminary Definitions

The terminology of this section is adapted from standard usage in the lambda
calculus, and appears e.g in Curry[12].

The relation > defined in each of the axiom systems presented here is a
monotone relation, i.e. it has the following properties:

   Reflexivity: For every X, X>X.

   Transitivity: If X>Y and Y>Z, then X>Z.

   Monotonicity: If X>Y and B is the result of substituting X for an occurrence
        of Y in expression A, then B>A.  X for an occurrence of Y, then B>A.

The above properties are assumed to be axioms of each system.

Certain of the axioms to be presented lead to a distinction between the
operations of contraction and abstraction; for example, the derivation of
S[A;x;M][1] from ((LAMBDA(x)M)A), justified by the beta axiom of the lambda
calculus, may be termed a beta-contraction. The inverse operation of
converting S[A;x;M] to ((LAMBDA(x)M)A) may be termed a beta-abstraction. An
expression which is a candidate for contraction is called a redex; thus
((LAMBDA(x)M)A) is a beta-redex in the lambda calculus. The result of
performing a contraction on a redex X is termed the contractum of X.

An expression in a particular calculus is in normal form if it contains no

---

[1] Recall that S is the substitution operation of the lambda calculus, Defn
[2.6].

redex applicable to that calculus. We say further that the expression X is in beta-<u>normal</u> <u>form</u> if X contains no beta-redex, and similarly for the delta, *, and E redexes to be defined presently. The statement that X is in normal form, without further qualification, may be taken to mean that X contains no beta-, delta-, *-,, or E-redexes.

We shall often use the notation X{Y} to designate an expression X containing a particular instance of a subexpression Y; having identified an expression with the notation X{Y}, we shall then use an expression of the form X{Z} to denote the result of replacing Y in X{Y} by the expression Z. In this notation, the monotonicity of $\geqslant$ is the implication of X{Y}$\geqslant$X{Z} by Y$\geqslant$Z.

A relationship of the form A$\geqslant$B is in general derived through a series of steps $A_1 \geqslant A_2$, $A_2 \geqslant A_3$, where each $A_i \geqslant A_{i+1}$ involves the substitution of an expression Y′ in $A_i$ for an occurrence of an expression Y$\geqslant$Y′. The monotonicity of $\geqslant$ justifies each such substitution, and the transitivity assures that the validity of the entire series follows from the validity of the individual steps. We shall use the terminology

Defn 4.1: A <u>reduction</u> <u>step</u> in A from X to Y, for expressions X and Y and a particular axiom system A, is a proof that X$\geqslant$Y by a single application of an axiom of A.

Defn 4.2: A <u>reduction</u> <u>sequence</u> from $X_0$ to $X_n$ in system A is a series $X_0 \geqslant X_1 \geqslant ... \geqslant X_n$ such that each $X_i \geqslant X_{i+1}$ is a reduction step in A.

## 4.2: The Either-R Theories

The first axiom, common to each of the systems presented, is taken directly from the lambda calculus:

Axiom <u>alpha</u>: (Renaming) Let E be an expression of the form (LAMBDA(X)A) where X is any variable and A is an expression, and let Y be any variable not occurring free in A. Then E=(LAMBDA(Y)S[Y;X;A]).

We say that expressions A and B are __congruent__ if A can be converted to B by alpha conversion alone. Congruence is thus reflexive, symmetric and transitive, and to simplify subsequent proofs we shall often allow ourselves to treat congruent expressions as identical.

The next axiom is a restricted form of the beta axiom of the lambda calculus, allowing beta conversion only on a beta-redex whose argument is in normal form:

Axiom __beta-R__: (lambda conversion) Let E be an expression of the form ((LAMBDA($\underline{a}$)$\underline{b}$)$\underline{c}$) where $\underline{c}$ is in normal form. Then E=E´, where E´ is the contractum S[$\underline{c}$;$\underline{a}$;$\underline{b}$] of E.

The following axiom provides a paradigm for delta-conversion, the application of primitive functions to arguments in normal form. A particular calculus will have a family of delta rules, specifying the behavior of each primitive -- e.g. the delta rule for the primitive PLUS asserting the equivalence of (PLUS n m) to n+m for all integers n and m. Of interest here is the general form of such rules:

Axiom __delta__: Let E be an expression of the form (A B) where A is a primitive function symbol and B is a normal form expression containing no free variables. Then E=E´, where E´ is the contractum of E derived from B by the (here unspecified) rules associated with A.

We may term such an expression E a __delta-redex__, and the conversion of E to E´ is a __delta-contraction__. Since the relation between E and E´ is equivalence, the axiom provides also for the __delta-abstraction__ of E´ to E.

We note that axioms alpha, beta-R, and delta define a lambda calculus under the equivalence relation =; no use has been made of the asymmetric relation >.

We shall term an expression of the form (EITHER $a_1$ $a_2$), where $a_1$ and $a_2$ are arbitrary expressions, an __E-redex__. We treat the E-redex as a new syntactic construct, rather than attempting to classify EITHER as an added primitive function whose operation is specified by delta rules. In particular, we regard the restriction that arguments of primitive functions be in normal form

as unacceptable to the process of EITHER-conversion.

Axiom epsilon: (EITHER-contraction): If E is an expression of the form (EITHER
$a_1$ $a_2$) where $a_1$ and $a_2$ are expressions, then E$\gg a_1$ and E$\gg a_2$.

Axiom mu: For every expression E, E=(EITHER E E).

Axiom rho: (EITHER-distribution) If E is an expression of the form (f (EITHER
a b)), where f, a, and b are arbitary expressions, then E=E´ where E´ is
the expression (EITHER (f a)(f b)).

The conversion of the redex (EITHER $a_1$ $a_2$) to one of the expressions $a_1$ or $a_2$
will be termed an E-contraction.  The conversion of an expression E to (EITHER
E E) will be called an E-abstraction.

### 4.2.1:  Properties of Either Theories

The elementary relationships established in this section hold for subequent
theories as well as for Either-R.  In addition to their usefulness in proofs,
they provide a preliminary reassurance that the Either-R axioms are consistent
with the intuitive semantics of EITHER.

Thm 4.3:  X$\gg$Y if and only if, for all Z,

$$Y\gg Z \implies X\gg Z$$

proof: only if: by the transitivity of $\gg$.
   if: Let Z be Y; then Y$\gg$Y by the reflexivity of $\gg$, hence X$\gg$Y by above
   hypothesis.

The above theorem is consistent with the intuitive notion that X$\gg$Y means
values derivable from Y are also derivable from X.

Axiom mu justifies the trivial abstraction of an expression E to the
expression (EITHER E E);  The following theorem shows that nontrivial EITHER
expressions may be abstracted:

Thm 4.4:  Let X, A, and B be expressions such that X≯A and X≯B.  Then
        X≯(EITHER A B).

  proof:  By Axiom mu, X≯(EITHER X X).
        But since X≯A and X≯B, (EITHER X X)≯(EITHER A B) by the monotonicity of
        ≯.  Hence X≯(EITHER A B).


We may apply this theorem, for example, to the expression A given by
                A≡((LAMBDA(X)(PLUS X 3))(PLUS 1 2))
By performing single beta and delta contractions, repectively, on A we deduce
the relations
                        A≯(PLUS (PLUS 1 2) 3)
                      A≯((LAMBDA(X)(PLUS X 3)) 3)
Application of Thm 4.4 yields the result
                A≯(EITHER (PLUS (PLUS 1 2) 3)((LAMBDA(X(PLUS X 3)) 3))
This demonstrates that the Either-R theory allows EITHER-free expressions
(such as A above) to be converted to expressions containing EITHER.


Thm 4.5:  X≈Y if and only if for all Z, X≯Z<=>Y≯Z.

  proof:  is by two applications of 4.3.



Thm 4.6:  For all f, g, and a,
        ((EITHER f g) a)≯(EITHER (f a)(g a))

  proof:  By Axiom epsilon, ((EITHER f g) a)≯(f a) and ((EITHER f g) a)≯(g a);
        hence, by Thm 4.4, ((EITHER f g) a)≯(EITHER (f a)(g a)).


The intuitive arguments of the last chapter suggest that the above result
could be strengthened to full equivalence (i.e., ≈), and this more powerful
result may in fact be a theorem in our Either theories; however we have not
pursued this equivalence since it is irrelevent to the subsequent proofs.

4.2.2:  EITHER and Evaluation Order

Chapter 2 notes the distinction between normal and applicative order
evaluation, characteristic respectively of the N and T interpreters.
Applicative order evaluation, in which arguments to a function are evaluated
prior to the application of the function, is shown in that chapter to lead to
the inexpressiblity of certain functions which ignore their arguments.  For
example, the applicative order evaluation of the expression

$$((LAMBDA(X)3)\ A)$$

does not terminate if the value of A is undefined, whereas the normal order
evaluation of that expression yields the value 3.

The restricted conversion of the beta-R axiom is similar to applicative order
evaluation -- in each case, the argument to a function must be avaluated
(reduced to normal form) before the application of the function (beta
conversion).  The only distinction between beta-R conversion and the
applicative order of the T interpreter is the degree of evaluation required;
while Either-R requires that arguments be reduced to normal form, T requires
only that they be reduced to lambda expressions or atoms.  We may thus, view
the restriction on beta conversion as a more serious constraint than the
applicative order evaluation of T.

The motivation for this restriction in the Either-R system is our intuitively
based demand that the axiom of EITHER-distribution, rho, hold.  This axiom is
in fact inconsistent with the unrestricted beta conversion of the lambda
calculus; consider, for example, the expressions I, Z, and F defined by

$$I\ \equiv\ (LAMBDA(X)X)$$
$$Z\ \equiv\ (LAMBDA(Y)(LAMBDA(X)X))$$
$$F\ \equiv\ (LAMBDA(H)(H\ H))$$

Using the axioms of Either-R (notably EITHER distribution) in conjunction with
unrestricted beta conversion, we may deduce that I≡Z as follows:  By Axiom mu,

$$I\ \equiv\ (EITHER\ I\ I)$$

and by (restricted) beta abstraction on each of the terms of the E-redex,

$$I\ \equiv\ (EITHER\ (F\ I)(F\ Z))$$

since both (F I)$\equiv$I and (F Z)$\equiv$I.  Then the axiom of EITHER distribution yields

$$I \;\equiv\; (F \;(EITHER \;I \;Z))$$

from which, using unrestricted beta conversion (as the argument is an E-redex and hence not in normal form) we deduce that

$$I \;\equiv\; (((EITHER \;I \;Z)(EITHER \;I \;Z))$$

whence by EITHER contraction

$$I \;\succ\; (I \;Z) \;\equiv\; Z$$

Thus we have derived I$\succ$Z; to show Z$\succ$I (and hence I$\bullet$Z) we make the deductions

$$I \;\succ\; Z$$
$$(I \;Z) \;\succ\; (Z \;Z)$$
$$Z \;\succ\; I$$

using the monotonicity of $\succ$ and beta-R abstraction.

It follows that, using unrestricted beta conversion in conjunction with the Either-R axioms, we can prove every pair of expressions equivalent -- i.e., the system is inconsistent.  We avoid this pitfall in Either-R by means of the restriction on beta conversion.  The beta-R restriction is not, however, the only solution to this problem, and in Chapter 7 an alternative axiom system -- designated the Either-K theory -- is presented.

It should be noted at this point that the restriction on beta conversion is expensive in terms of expressive power.  It prohibits, for example, the reduction of the expression

$$((LAMBDA(X)3) \;((LAMBDA(Y)(Y \;Y))(LAMBDA(Y)(Y \;Y)))$$

to the value 3, since the argument in that expression has no normal form.  A more serious drawback is that it interferes with the expressibility of recursive functions since recursion requires, in the lambda calculus, the application of functions to arguments having no normal forms.  Chapter 5 is devoted to the mechanism of *-conversion, which mitigate these limitations imposed by the restricted beta conversion.

4.2.3:  Consistency of Either-R

An extension of the axiomatic basis of the Lambda calculus may lead to
inconsistencies, e.g. the equivalence of 1 and 2.  Such equivalences do not
hold in the conventional lambda calculus;  in particular, the first Theorem of
Church and Rosser establishes the consistency of the Lambda Calculus axioms by
showing that the proposition X=Y is not provable for any pair of expressions X
and Y having incongruent normal forms.  We are thereby assured that the
equivalence relation = established by the lambda calculus does not place every
expression in a single equivalence class, and thus that the cardinality of the
domain of the Lambda Calculus is greater than 1.  The existence of infinite
sets of mutually incongruent normal forms[1] shows that the domain of the lambda
calculus is infinite.  Moreover, an important theorem of Boehm[20] shows that
any axiomatic assertion of the form X=Y, where X and Y are incongruent normal
forms, leads to an inconsistency.

The theorems of Church-Rosser and Boehm are, not surprisingly, inapplicable to
the axiomatic extension presented here.  Furthermore, they probably cannot be
augmented in minor ways to argue the consistency of the present system, as the
uniqueness of normal forms, on which they depend, has been compromised by our
extension.

Accordingly, is the purpose of this section to establish that the domain of
the lambda calculus is a subset of the domain of the Either-R system, and that
the new equivalence relation • is consistent with the relation = of the lambda
calculus.  In particular we wish to show that, for any two either-free
expressions X and Y, X=Y if X•Y.  Proof of this assertion establishes that

1) The domain of the Either-R system includes the domain of the lambda
   calculus, hence the new system is nontrivial (having infinite
   cardinality); and

2) The semantic equivalence defined by the Either-R calculus, applied to
   EITHER-free expressions, is a subset of the equivalence of the lambda
   calculus.

---

[1] For example, the set I≡(lambda(x)x), I´≡(lambda(x)I), I´´≡(lambda(x)I´),
etc.

It has been noted that in the Either-R system there are expressions X and Y
such that X=Y but for which X≠Y is not provable -- a consequence of the
restriction on beta conversion which is explored further in the analysis of
the R-* system in the following chapter.

We procede to the consistency proof, beginning with with the following
definition:

Defn 4.7:    The **EITHER-free** expression X′ is an **e-residue** of the expression X
if and only if X′ may be derived from X by replacing every e-redex
(EITHER $x_1$ $x_2$) in X by one of the operands $x_1$ or $x_2$.

Thus the expression X′ is an e-residue of X if X′ is EITHER-free and X⊃X′ may
be demonstrated solely by means of EITHER-contraction (axiom epsilon).

Defn 4.8:    The expression X is **unitary** if and only if there exists some
EITHER-free expression Y such that, for every e-residue X′ of X, X′=Y (in
the lambda calculus).

Thus
                    (EITHER (LAMBDA(X)X) (LAMBDA(Y)Y))
is unitary, since its e-residues (LAMBDA(X)X) and (LAMBDA(Y)Y) are congruent.
We note that EITHER-free expressions are unitary, although unitary expressions
are not necessarily EITHER-free, as the above example demonstrates.
Furthermore, a unitary expression X may contain subexpressions which are not
unitary; witness the expression

                ((LAMBDA(X)(DIFFERENCE X X))(EITHER 2 3))                [4.9]

whose e-residues are

                    ((LAMBDA(X)(DIFFERENCE X X)) 2)
and
                    ((LAMBDA(X)(DIFFERENCE X X)) 3)

each of which is convertible to 0 by the rules of the Either-R system.  Hence
expression [4.9] is unitary;  it contains, however, the subexpression
                        (EITHER 2 3)

which has e-residues 2 and 3, which are not equivalent under =. Hence the subexpression is not unitary.

The proof of the consistency of Either-R is based on the observation that, while EITHER may be introduced into EITHER-free expressions by EITHER-abstraction, the result is necessarily unitary. Moreover, the axioms of Either-R preserve the unitary nature of expressions; we will thus argue that the result of an Either-R reduction sequence on an EITHER-free expression must be unitary. We now introduce a relation which orders expressions by the interconvertability, in the lambda calculus, of their e-residues:

Defn 4.10: For any expressions X and Y we say that X __encloses__ Y if, for every e-residue $Y'$ of Y, there is an e-residue $X'$ of X such that $X'=Y'$ in the lambda calculus.

Observe that enclosure is reflexive and transitive; the following lemma establishes that it is monotonic:

Lemma 4.11: Let Y be a subexpression of X{Y} and let Y enclose Z. Then X{Y} encloses X{Z}.

   proof: Each e-residue of X{Z} is of the form $X'\{Z'\}$ where $Z'$ is an e-residue of Z; and for each e-residue $Y'$ of Y there is a corresponding e-residue $X'\{Y'\}$ of X{Y}. Hence for each e-residue $X'\{Z'\}$ of X{Z} there is an e-residue $X'\{Y'\}$ of X{Y} such that $Y'=Z'$; it follows that $X'\{Y'\}=X'\{Z'\}$ hence X{Y} encloses X{Z}.

Corollary 4.12: If X{Y} is unitary and Y encloses Z, then X{Z} is unitary and every e-residue of X{Z} is convertible to an e-residue of X{Y}.

Lemma 4.13: Let X→Y be a single reduction step in Either-R. Then X encloses Y.

proof: Let U be the subexpression of X which is replaced by an expression W
in the reduction step X→Y. By Lemma 4.11, we need only to show that U
encloses W to establish that X encloses Y. We exhaustively examine the
possible reduction steps from U to W:

Case 1: Alpha conversion on U. Then U and W are congruent, and for each
e-residue W´ of W there is a congruent e-residue U´ of U.

Case 2: beta-R conversion on U. Let P be a beta-redex of the form
((LAMBDA(X)M{X})A) where A is in normal form, and let Q be the contractum
S[A;X;M{X}] of P. Then every e-residue P´ of P is of the form
((LAMBDA(X)M´{X})A) where M´{X} is an e-residue of M{X}, and there is one
such e-residue P´ for every e-residue M´ of M. Each e-residue W´ of W is
of the form M´{A} and there is one such e-residue W´ for each e-residue M´
of M. For each M´ the corresponding e-residues of P and Q are
((LAMBDA(X)M´{X})A) and M´{A} respectively, which are interconvertible in
the lambda calculus by a single beta conversion; hence P encloses Q and Q
encloses P. W is either a beta-R contraction or a beta-R abstraction of U,
hence U encloses W.

Case 3: delta-conversion on U. If either U or W is a delta redex, then both U
and W are EITHER-free and thus U encloses W.

Case 4: EITHER contraction. If U is an expression of the form (EITHER $A_1$ $A_2$),
clearly U encloses both $A_1$ and $A_2$; each e-residue of W is an e-residue of
$A_1$ or of $A_2$.

Case 5: EITHER-abstraction. Then W is of the form (EITHER U U), and each
e-residue of W is an e-residue U´ of U.

Case 6: EITHER-distribution. Let P be an expression of the form
(EITHER (F A)(F B))
and let Q be
(F (EITHER A B))
The e-residues of P consist of all the expressions of the forms (F´ A´) and
(F´ B´) where F´, A´, and B´ are respectively e-residues of F, A, and B.
We note that the e-residues of Q consist of exactly the same set of
expressions, hence P encloses Q and Q encloses P. Thus for a conversion

U>W of the forms P>Q or Q>P, U encloses W.

This completes the proof of Lemma 4.13.


We present the obvious generalization of this result as


Corollary 4.14:  Let X and Y be expressions such that X>Y in the Either-R
        system.  Then X encloses Y.

    proof follows directly from Lemma 4.13 and the transitivity of the enclosure
        relation.


This corollary shows that the ordering > of the Either-R system implies
enclosure; thus the number of distinct (under = of the lambda calculus)
e-residues of an expression X can only be <u>decreased</u> by a reduction step in
Either-R.  While each reduction step may introduce new E-redexes (by
EITHER-abstraction), the terms of each redex so introduced are necessarily
interconvertable.  The consistency of the Either-R theories is a special case
of this corollary:


Thm 4.15:  Let X and Y be EITHER-free expressions such that X>Y in the
        Either-R theories.  Then X=Y in the lambda calculus.

    proof: By Corollary 4.14, X encloses Y; since X and Y are each EITHER-free,
        X and Y are respectively e-residues of X and Y.  Hence X=Y in the lambda
        calculus.


The above theorem establishes that the Either-R theories are consistent in the
sense that they introduce no new equivalences between expressions which are
distinct in the lambda calculus; and are hence of infinite cardinality.  It is
noteworthy at this point that the above proof, specifically Lemma 4.13,
depends on our restriction on beta conversion.  when unrestricted beta
conversion is allowed (as in the Either-K theories presented in Chapter 7) it
is not true in general that every beta-redex X encloses its contractum X´, as
demonstrated by the beta redex

$$A \underline{=} ((LAMBDA(X)(PLUS\ X\ X))(EITHER\ 2\ 3))$$

whose e-residues are each convertible to 2 and 3, respectively, while the contractum of A

$$(PLUS\ (EITHER\ 2\ 3)(EITHER\ 2\ 3))$$

has an e-residue (PLUS 2 3) which is convertible neither to 2 nor to 3.

### 4.3:  Summary

This chapter defines the ground rules for the axiomatization of Either theories and presents the Either-R theory. While the usefulness of this system is limited due to the restriction placed on beta conversion, it develops much of the mechanism to be used in subsequent chapters.

The principal distinction to be made between the Either theories lies in the circumstances in which beta-conversion is allowed. The Either-R Theories, which prohibit beta-conversion unless the argument to be substituted is in normal form, allow the distribution of functions over the terms of an EITHERexpression - a relationship which we find intuitively gratifying. Unfortunately this restricted beta-conversion results in a very weak theory, a problem to which the next chapter is devoted.

The Either-R theory presented in this chapter is shown to be consistent in the sense that X$\rightarrow$Y, where $\rightarrow$ is the ordering defined by the new axioms, is not a tautology. The proof is based on the consistency of the lambda calculus; specifically, it is shown that, for expressions X and Y which are EITHER-free (and thus admissible syntactically in the lambda calculus) X$\rightarrow$Y implies the interconvertability of X and Y. This general technique will be followed in subsequent consistency proofs as well.

## Chapter 5:
## *-Conversion

It was noted in the previous chapter that the restricted lambda conversion of the beta-R axiom, i.e. the requirement that the argument of a beta-redex be in normal form before the contraction of that redex, severely limits the expressive power of languages based on the Either-R theory. In particular, the inexpressibility of recursive functions constitutes an intolerable restriction since it renders such languages functionally incomplete.

The mechanism of *-conversion, to be introduced in the present chapter, ameliorates this limitation by extending the ordering relation $\geqslant$ in a way which is consistent with its function in the Either-R theory. Although *-conversion and EITHER reduction are in an important sense complementary operations, their respective mechanics may be dealt with separately; thus for the purposes of this chapter we temporarily disregard the axioms of EITHER conversion. In Chapter 6 we combine the two mechanisms.

The semantic interpretation of $\geqslant$ suggested by the Either-R theory is one of inclusion of values; it was noted that $X \geqslant Y$ signifies, in general, that each value of Y is also a value of X. The corresponding relation in the semantic domain $F^*$ is set theoretic inclusion. Thus if x and y are the semantic elements of $F^*$ corresponding to X and Y, respectively, then $X \geqslant Y$ implies that y is a subset of x. Consistent with the semantic notions of Chapter 3, the expression (EITHER X Y) corresponds in $F^*$ to the union of the elements x and y. It was further suggested that the undefined computation corresponds, in $F^*$, to the empty set -- i.e., it has no values whatsoever.

This chapter develops the syntactic analog of the empty set in $F^*$. Specifically, the new syntactic element * is introduced as the canonical normal form representation of the undefined computation. The interpretation of $\geqslant$ as set theoretic inclusion in $F^*$ suggests that, for _every_ expression X, $X \geqslant *$ (since every set has the empty subset). It would seem, then, that the consummation of the semantics of EITHER reduction requires that its syntactic mechanism reflect this aspect of the structure of $F^*$.

5.1:   The R-* Theories

We now focus our attention on *-conversion and its relation to the restricted
beta conversion.  To this end we consider the R-* system whose axioms include
alpha, beta-R, and delta discussed previously, in addition to the following:

Axiom sigma: (*-contraction): For every expression E, E>*.

Thus * is an expression in the R-* system which is lower, in the sense of >,
than every other expression.  While every expression is reducible to *, * is
itself only reducible to * (as * is not a beta- or delta-redex, and contains
no variables).

Defn 5.1:   An expression of the form (* A), where A is an arbitrary
       expression, is called a *-redex.

Consistent with our previously defined notion of normal forms, we shall
henceforth require an expression X to contain no *-redexes if it is in normal
form.  Noting that the only conversion which may be performed on a *-redex
without resulting in another *-redex is its replacement by *, we shall say
that the contractum of a *-redex is *.

5.1.1:   Significance of normal forms

The restricted lambda conversion allowed by the beta-R axiom bears a curious
resemblance to the lambda-I calculi of Church[1].  In these systems, Church
specifically prohibits expressions of the form (LAMBDA(X)M) unless the
variable X appears free in the body M; thus the lambda-I systems exclude, in
general, functions which ignore their arguments.  A principal consequence of
this restriction is the fact that, for expression X to have a normal form,
every subexpression of X must have a normal form.  We note, with passing
interest, that the normal form restriction of beta-R allows us to derive any
normal form in the lambda-I calculus which is possible using unrestricted beta
conversion;  this follows from the observation that in the lambda-I system we
can always reduce the argument in a beta-redex to normal form before
contracting the redex.

Church's preference for the lambda-I over the unrestricted "lambda-K"[1]
theories stems from the elusive nature of those expressions having no normal
forms.  The theorem of Boehm assures us that expressions having incongruent
normal forms are semantically distinct, and the theorems of Church-Rosser
guarantee that equivalences between expressions having normal forms are
decidable.  The semantics of normal forms is consequently uncomplicated:
every pair of semantically equivalent normal form expressions is provably
equivalent, and for every pair of incongruent normal forms we can find a
context in which they produce different values.

The admission of expressions having no normal forms compromises this situation
severely.  The requirement that a semantic equivalence relation be
extensional, i.e. that equivalent expressions produce equivalent values in
identical contexts, leads to a distinction between semantic equivalence and
the equivalence of interconvertability under the lambda calculus.  Scott[22],
for example, demonstrates an infinite sequence $Y_0$, $Y_1$, ... of fixed point
operators which are not convertible to one another despite the fact that they
produce the same values when embedded in identical contexts.  The problem of
constructing a functional domain for the lambda calculus is fundamentally
equivalent to the definition of an extensional relation of semantic equialence
over the expressions of that calculus, a project whose recent success is due
to Scott.  The technique used by Scott[5,6,22] involves the notion of
successively better approximations to the abstract semantic element
represented by an expression X, so that the semantic element associated with X
becomes the limit of this sequence of approximations.  In the Scott model, a
function f' approximates every extension f of f';  more generally, f'
approximates f if and only if for every z, f'[z] approximates f[z].  This
notion of approximation seems essential to the interpretation of domain
elements as functions, largely because the theories of functions with which we
are familiar employ type restrictions ruling out self-application.[2]

---

[1] Church[1] and Curry[12]  refer to the unrestricted conversions of the
conventional lambda calculus as lambda-K conversion, presumably because of the
admissibility  of the combinator K=(LAMBDA(X)(LAMBDA(Y)X)) in these systems.
K is excluded from the restricted lambda-I systems by the non-occurence of the
bound variable Y in the body of (LAMBDA(Y)X).

[2] In particular, (LAMBDA(X)(X X)) is difficult to interpret as a function in
the usual set-theoretic way.  Hindley[21] speculates that a theory of
functions based on combinatory logic, rather than set theory, might
consistently allow self-application;  while awaiting further developments we
remain pessimistic.

The mechanism of *-conversion presented in this chapter is reminiscent of the
Scott construction. Specifically, we introduce means by which the various
approximations of an abstract semantic element can be represented as
expressions in the language itself, and provide for the syntactic conversion
of an element X to an approximation X´ of X. We have thus come to view
*-conversion as a syntactic analog of the Scott construction in which
approximations are expressed in the domain of the language rather than in the
abstract semantic domain.

The addition of *-conversion to the lambda calculus leads to a multiplicity of
normal forms for every expression. We shall see, for example, that the Y
operator

$$Y \equiv (LAMBDA(F)((LAMBDA(H)(F(H\ H)))(LAMBDA(H)(F(H\ H)))))$$

which has no normal form in the conventional lambda calculus, has infinitely
many normal forms

$$*$$

$$(LAMBDA(F)(F\ *))$$
$$(LAMBDA(F)(F\ (F\ *)))$$
$$(LAMBDA(F)(F\ (F\ (F\ *))))$$

$$-\ -\ -$$

when *-conversion is admitted. Each of these normal forms may be interpreted
as an approximation to the Y operator, and in any context where Y gives a
normal form value, one of the above normal forms of Y will give an identical
value. Since the semantic element associated with each of these normal forms
is clear (in the sense that normal forms are semantically distinct) we retain
something of the semantic simplicity of the lambda-I calculus. The semantic
value of a given expression is simply the set of normal form values of that
expression, and expressions X and Y are semantically equivalent if and only if
they have identical sets of normal forms.

One of the motivations for *-conversion is to enable us to retain the power of
the unrestricted (lambda-K) calculus while restricting beta conversion. It is
intuitively reasonable to expect that one can always find a sufficiently close
approximation to the argument of a lambda expression that the restriction on

beta conversion becomes unimportant where *-conversion is allowed, and much of the remainder of this chapter is devoted to the proof that this is in fact the case.


### 5.1.2: Theorem on Normal Forms

The main result of this section sheds light on the ordering (under $\geqslant$) of the normal forms derivable in R-* from an expression A. We begin with the following definition, adapted from Curry[12]:

Defn 5.2: Let P be a redex and Q be a subexpression in an expression B, and let B' be the result of replacing P by its contractum P' in B. We define the residuals of Q with respect to P as subexpressions of B' designated as follows:

Case 1: P and Q are the same redex in B. Then Q has no residual with respect to P.

Case 2: P and Q are non-overlapping subexpressions of B. Then the residual Q' of Q is that subexpression in B' which is homologous[1] to Q in B.

Case 3: P is a subexpression of Q. Then the residual of Q in B' is the expression Q' which is homologous to Q in B. We note that the occurrence of P in Q has been replaced by P' to make Q'.

Case 4: P is a beta-redex ((LAMBDA(X)M)A), and Q is a subexpression of A. Then P' is S[A;X;M] and contains n instances of A corresponding to the n free occurrences of the variable X in M; let these instances of A be identified as $A_1$ ... $A_n$. Each Ai contains an instance Qi of the redex Q; these n expressions $Q_1$ ... $Q_n$ are the n residuals of Q in B'. Note that n may be zero, in which case we term the contraction of P a cancellation and Q has no residuals.

---

[1] homologous subexpressions occupy the same relative position in their containing expressions; thus A in ((X (W A) Z) Y) is homologous to B in ((P (Q B) R) S) independently of the structure of the subexpressions X, W, Z, Y, P, Q, R, and S.

Case 5: P is a beta-redex ((LAMBDA(X)M)A) and Q is a subexpression of M. Then P´ is S[A;X;M] and the residual Q´ of Q is the subexpression of P´ which is homologous to Q in M.

Case 6: P is not a beta-redex, and Q is a subexpression of P. Then Q has no residual in B´.

Informally, a residual of an expression Q is an image of Q after a contraction. Consider, for example, the residuals of the subexpression (PLUS 3 4) in the beta-redex

$$((LAMBDA(X)(PLUS X X))(PLUS 3 4))$$ [5.3]

whose contractum is the expression

$$(PLUS (PLUS 3 4)(PLUS 3 4))$$

We note that the two residuals of the subexpression (PLUS 3 4) of expression [5.3] are the occurences of (PLUS 3 4) in the contractum. Contraction in the delta redex (PLUS 3 4) in expression [5.3] yields the residual

$$((LAMBDA(X)(PLUS X X)) 7)$$

We shall occasionally find it useful to speak of the residual of an expression Q after a series of contractions; we may thus refer to $Q_n$ as a residal of Q with respect to the sequence of contractions $B \rightarrow B_1 \rightarrow \ldots \rightarrow B_n$ if there is a subexpression $Q_{n-1}$ of $B_n-1$ such that $Q_{n-1}$ is a residual of Q and $Q_n$ is a residual of $Q_{n-1}$. Thus consecutive beta- and delta-contractions on expression [5.3] yield

$$(PLUS 7 (PLUS 3 4))$$

which contains a single residual of the subexpression (PLUS 3 4). The following lemma establishes that the residual of a redex is always a redex:

Lemma 5.4: Let P and Q be redexes in an expression B, and let Q´ be a residual of Q with respect to P. Then Q´ is a redex.

proof: We consider the following collectively exhaustive cases:

Case 1: P and Q are non-overlapping.[1]  Then Q′ is the same redex as Q.

Case 2: P is a subexpression of Q; we consider the cases of the syntax of Q:

   a) Q is a beta-redex of the form ((LAMBDA(X)M A).  If P is a subexpression of M, then Q′ is the beta-redex ((LAMBDA(X)M′)A).  If P is a subexpression of A, then Q′ is the beta-redex ((LAMBDA(X)M)A′).

   b) Q is a *-redex of the form (* M); then P must be a subexpression of M, and Q′ is the *-redex (* M′).

   c) Q cannot be a delta-redex, as it contain P.

Case 3: Q is a subexpression of P; we consider cases of the syntax of P:

   a) P cannot be a delta-redex, as it contains the redex P.

   b) P cannot be a *-redex, as then Q would have no residual.

   c) P is a beta-redex of the form ((LAMBDA(X)M)A) where Q is a subexpression of A.  If Q is cancelled by the contraction of P, then Q has no residual; hence M must contain 1 or more free occurrences of X.  Then each residual of Q is the redex Q itself.

   d) P is a beta-redex ((LAMBDA(X)M)A) where Q is a subexpression of M. We examine syntactic cases of Q:

      i) Q is a delta-redex; then Q′ is identical to Q, since Q may contain no free variables (in particular, no free occurrence of X).

      ii) Q is a *-redex (* M).  Then Q′ is the *-redex (* M′).

      iii) Q is a beta-redex ((LAMBDA(Y)B)C).  Then Q′ is a beta-redex of the form ((LAMBDA(Y)B′)C′).

---

[1] Two expressions are non-overlapping if neither is a subexpression of the other.

The converse of the above lemma is not in general true, i.e., the residual P′
of P may be a redex even though P is not.  Consider for example the expression

$$P \equiv (((LAMBDA(X)(LAMBDA(Y)Y))\ 3)\ 4)$$

which is not a redex.  Contraction of the beta-redex in P yields the residual
P′ of P given by

$$P' \equiv ((LAMBDA(Y)Y)\ 4)$$

which is a beta-redex.

We should like to distinguish between reduction steps in R-* which are
contractions and those which are abstractions; for this distinction the
following notation is convenient:

Defn 5.5:   A __contraction step__ A≫B is a single reduction step from A to B
    which is either a beta-, delta-, or *-contraction.

Defn 5.6:   A __contraction sequence__ $A_0 \gg A_1 \gg \ldots \gg A_n$ from $A_0$ to $A_n$ is a reduction
    sequence from $A_0$ to $A_n$ containing only alpha-conversions and contraction
    steps.  The length n of such a sequence is the number of contraction
    steps in the sequence.

We now examine contraction sequences which terminate in normal forms,
beginning with

Lemma 5.7:   Let X{Y} be an expression containing a redex Y, and let
    X{Y}≫...≫X′ be a contraction sequence of length n, where X′ is in
    normal form.  Then there is a contraction sequence X{Y′}≫...≫X′, where
    Y′ is the contractum of Y, of n or fewer steps.

   __proof__ is by induction on n.

       __basis__ n=1:  X′ contains no redex, hence Y must be either contracted or
       cancelled (by a beta- or *-contraction).  If Y is contracted then
       X[Y′]≫X′ by the null sequence.  If Y is cancelled then X[Y′]≫X′ by the
       same contraction as X[Y]≫X′.

induction: We assume the lemma to be true for sequences containing n or fewer steps. Consider the first contraction step $X[Y] \gg X_1$ in the n+1-step sequence $X[Y] \gg \ldots \gg X'$, and let $Y_1 \ldots Y_j$ be the j residuals of Y in $X_1$. If j=0 then the argument in the basis applies, as Y is either contracted or cancelled in the first step. If j>0, j applications of the induction hypothesis establish that $X_1' \gg \ldots \gg X'$ in n-1 or fewer steps, where $X_1'$ is the result of contracting each $Y_i$ in $X_1$. But $X[Y'] \gg X_1'$ in a single step; hence $X[Y'] \gg X'$ in n or fewer steps.


The significance of Lemma 5.7 is that the contraction of a redex Y in expression X cannot prolong the reduction of X to normal form. Informally, we expect that if the subexpression Y plays a significant role in the evaluation of X, the contraction of Y will shorten the reduction of X; if, however, Y is irrelevent to the value of X then Y may be replaced by an arbitrary expression with no effect on the evaluation of X. This consideration motivates

Lemma 5.8: Let $B_0 \gg B_1 \gg \ldots \gg B_n$ be a contraction sequence of length n, and let $B_n$ be in normal form. Let P be a redex in $B_0$, and let P' be the contractum of P. Then one of the following applies:

a) There is a contraction sequence $B^* \gg \ldots \gg B_n$ of n or fewer steps, where $B^*$ is the result of substituting * for P in $B_0$; or

b) There is a contraction sequence $B' \gg \ldots \gg B_n$ containing fewer than n contraction steps, where B' is the result of replacing P in B by P'.

proof is by induction on the length n of the contraction sequence $B_0 \gg B_n$.

basis n=1; then $B_0 \gg B_n$ in a single contraction step. Let Q be the redex contracted in $B_0 \gg B_n$. If Q is the same redex as P, then B' is identical to $B_n$, and (b) is satisfied. Otherwise P must have no residual in $B_n$, since $B_n$ is in normal form and any residual of P is a redex. Then P must be cancelled by a beta- or *-contraction in $B_0 \gg B_n$, and (a) is satisfied.

induction: n>1. Consider the redex Q contracted in the step $B_0 \gg B_1$. If Q is the same redex as P, then (b) is satisfied as before. Otherwise we consider the j residuals $P_1 \ldots P_j$ of P in $B_1$. If j=0 then P is cancelled

in the step $B_0 \gg B_1$, and (a) applies. If $j > 0$, we apply (by the inductive
hypothesis) the lemma to the contraction sequence $B_1 \gg \ldots \gg B_n$, whose
length is $n-1$:

Case 1: Each residue Pi in $B_1$ is convertible to *; i.e., (a) applies to
each Pi. Then (a) applies to P in $B_0$, as $B* \gg B_1*$ in a single step,
where $B_1*$ is the result of replacing each Pi in $B_1$ by *.

Case 2: Some residue Pi of P in $B_1$ is not convertible to *; i.e., (b)
applies to Pi. By Lemma 5.7, contracting any $P_k$ in $B_1$ cannot prolong
the sequence $B_1 \gg \ldots \gg B_n$; by the induction hypothesis, there is at
least one $P_k$ whose contraction shortens the sequence. Then if $B_1'$ is
the result of contracting each $P_k$ in $B_1$, there is a contraction
sequence $B_1' \gg \ldots \gg B_n$ in fewer than $n-1$ steps. Since $B' \gg B_1'$ in a
single contraction step (of the same kind as $B_0 \gg B_1$) (b) is satisfied.

This completes the proof of Lemma 5.8.


The following theorem establishes a fundamental property of *-conversion.
Informally it ensures that, for any two normal form expressions $A_1*$ and $A_2*$
which are each derivable from an expression A, there is an expression $A*$ in
normal form which is an __upper__ __bound__ of $A_1*$ and $A_2*$ in the sense that $A* \gg A_1*$
and $A* \gg A_2*$, and furthermore that $A \gg A*$. This result is then extended to the
case of an arbitrarily large finite set of expressions $A_1* \ldots A_n$ each derivable
from A. The existence of normal form upper bounds of arbitrary sets of
expressions derivable from A is essentially equivalent to the proposition
that A can be approximated, to arbitrary accuracy, by normal forms derivable
from A.


Thm 5.9: Let $A_1*$ and $A_2*$ be normal form expressions and let A be any
expression such that $A \gg A_1*$ and $A \gg A_2*$. Then there exists an expression
$A*$ in normal form such that $A \gg A*$, $A* \gg A_1*$, and $A* \gg A_2*$.

__proof__: Let $P[n;m]$ be the proposition that Lemma 5.9 is true for every A,
$A_1*$, and $A_2*$ such that:
(i) $A \gg A_1*$ in $n_1$ steps and $A \gg A_2*$ in $n_2$ steps, where $n_1 + n_2 \leq n$; __and__
(ii) A contains m or fewer redexes.

Then the lemma is true if and only if $P[n;m]$ is true for all $n$ and $m$; we procede in the following steps:

1) For every $n$, $P[n;0]$ is true since in these cases $A$ contains no redex and is consequently in normal form.

2) For every $m$, $P[1;m]$ is true since in these cases either $A \equiv A_1{}^*$ or $A \equiv A_2{}^*$; hence $A$ must be in normal form and $A^* \equiv A$.

3) If for some $n$ and $m$ and for all $j$ $P[n,j]$ and $P[n+1;m]$ are true, then $P[n+1;m+1]$ is also true.

proof: Let $A$, $A_1{}^*$, and $A_2{}^*$ be expressions such that the premises of $P[n+1;m+1]$ are satisfied; then $A$ contains $m+1$ or fewer redexes, and $n_1+n_2 \geq n+1$ where $n_1$ and $n_2$ are the respective lengths of the sequences $A \gg A_1{}^*$ and $A \gg A_2{}^*$. We now choose an innermost redex $Y$ of $A$, i.e. a redex $Y$ which contains no other redex. Such a redex $Y$ must exist unless $A$ is in normal form, which is ruled out because $m+1 > 0$. Let $A\{Y\}$ denote $A$ (which contains $Y$ as a subexpression) and let $Y'$ be the contractum of the redex $Y$. Then by Lemma 5.8, one of the following applies:

a) $A\{*\} \gg A_1{}^*$ in $n_1$ or fewer steps, and $A\{*\} \gg A_2{}^*$ in $n_2$ or fewer steps.

b) $A\{Y'\} \gg A_1{}^*$ in $n_1'$ steps and $A\{Y'\} \gg A_2{}^*$ in $n_2'$ steps, where $n_1'+n_2' < n_1+n_2$.

If case (a) applies, then $A\{*\}$ has fewer than $m+1$ redexes, and by $P[n+1,m]$ the proposition $P[n+1,m+1]$ is true. If (b) applies, then $P[n+1,m+1]$ is true if $P[n;j]$ is true (where $j$ is the number of redexes contained in $A\{Y'\}$); by hypothesis, $P[n;j]$ is true for all $j$, hence $P[n+1;n+1]$ is true.

4) If for all $j$ $P[n;j]$ and $P[n+1;0]$ are true, then for all $i$ $P[n+1,i]$ is true.

proof is by induction on $i$. $P[n+1;0]$ follows directly from (1); $P[n+1;i+1]$ follows from (3) and $P[n+1;i]$.

5) For every i and j, P[i;j] is true.

> proof is by induction on i.
>
> basis: from (2), P[1,j] is true for all j.
>
> induction: Assume that P[i;j] is true for all j. By (1), P[i+1;0] is true; hence by (4), P[n+1;j] is true for all j.

This completes the proof of Theorem 5.9.


The proof of Theorem 5.9 involves a succession of steps from the expression A to the normal form $A^*$, such that the result $A_j$ of each step retains the property that $A_j \gg A_1^*$ and $A_j \gg A_2^*$. The moderate complexity of the proof stems from the obscure sense in which each step comes "closer" to $A^*$; by Lemma 5.8, each successive step from $A_j$ to $A_{j+1}$ either:

i) Reduces (by one) the number of redexes, while keeping the total number of steps in the contraction sequences $A_j \gg A_1^*$ and $A_j \gg A_2^*$ constant; or

ii) reduces the total number of contraction steps, while changing (increasing or decreasing) the number of redexes by some arbitrary finite amount.

The proof of Theorem 5.9 is essentially a demonstration that $A^*$ can always be derived from A by such a sequence in finitely many steps.

The generalization to arbitrary finite sets of normal forms follows naturally:


Corollary 5.10: Let A be any expression and let $A_1 \dots A_j$ be expressions in normal form such that, for each i, $A \gg Ai$. Then there exists an expression $A^*$ in normal form such that $A \gg A^*$ and, for each i, $A^* \gg Ai$.

> proof is by induction on j.
>
> basis: For j>2, the corollary is trivially true; for j=2, it is true by direct application of Theorem 5.9.
>
> induction: Assume the corollary is true for each set $A_1 \dots Ak$ containing fewer than j expressions. By Theorem 5.9, there is an expression $A_12^*$ in normal form such that $A_12^* \gg A_1$ and $A_12^* \gg A_2$ and $A \gg A_12^*$; by the induction hypothesis, we can now find an upper bound of the set $A_12^*, A3, \dots, A_j$

which contains j-1 expressions; let $A^*$ be the normal form upper bound of
this latter set. But, since $A^* \gg A_1 2^*$, it follows that $A^* \gg A_1$ and $A \gg A_2$;
hence for each Ai, $A^* \gg Ai$, and $A^*$ is the required upper bound.


The final theorem of this section establishes that, for the evaluation of any
particular expression X{Y} (i.e., the reduction of that expression to a normal
form) there exists a sufficiently good approximation $Y^*$ of Y such that $Y^*$ is
in normal form:


Thm 5.11:  Let $X\{Y\} \gg \ldots \gg X^*$ be a contraction sequence of length n, where $X^*$
       is in normal form. Then there exists an expression $Y^*$ in normal form,
       such that $Y \gg Y^*$ and $X\{Y^*\} \gg X^*$.

   proof is by induction on the length n of the contraction sequence. If n=0,
       then Y is in normal form and is the required $Y^*$. If n>0, we consider the
       residuals $Y_1 \ldots Y_j$ of Y in $X_1$. By the induction hypothesis each $Y_i$ can be
       contracted to a normal form $Y_i^*$, and the result $X_1^*$ of replacing each $Y_i$
       in $X_1$ by $Y_1^*$ is such that $X_1^* \gg X^*$. Since for each i $Y \gg Y_i^*$, by Corollary
       5.10 there is a $Y^*$ such that $Y \gg Y^*$ and for each i $Y^* \gg Y_i$. Then
       $X\{Y\} \gg X\{Y^*\} \gg X_1 \gg \ldots \gg X^*$.

We may speculate further on the structure of the set S of normal forms of an
expression A. The above theorem shows that any finite subset of S has an
upper bound in S;  since * is in S, we may claim further that each finite
subset in S has a lower bound in S. It seems likely that S forms a lattice
ordered by $\gg$, which is to say that each finite subset of S has both a least
upper bound and a greatest lower bound. In general such a lattice of normal
forms can be complete only for those expressions which have normal forms in
the lambda calculus.



### 5.1.3: Relation to the Lambda Calculus

In this section we demonstrate a sense in which the R-* theory is as powerful
as the (unrestricted) lambda calculus; in particular, we show that any
expression A which has the normal form A' in the lambda calculus has the same

normal form in R-*.


Thm 5.12:  Let $A_0 \to A_1 \to \ldots \to A_n$ be a sequence of beta- and delta-contractions
in the Lambda calculus (possibly intermixed with alpha conversions), and
let $A_n$ be in normal form.  Then $A_0 \gg A_n$ in R-*.

proof is by induction on n, the number of contractions in the sequence
$A_0 \to \ldots \to A_n$.

basis n=0; then $A_0$ and $A_n$ are identical, and the theorem is trivially
true.

induction: n>0; we assume then that $A_1 \gg A_n$ and must show that $A_0 \gg A_n$.  We
procede by showing that $A_0 \gg A_1$ for each of the possible contraction steps
$A_0 \to A_1$.  If the contraction step is an alpha- or delta- conversion, then
the same contraction can be performed in R-* hence $A_0 \gg A_1$; we thus need
only consider the case where $A_0 \to A_1$ by a beta contraction.  Let P be the
beta-redex contracted in the step $A_0 \to A_1$; then P is of the form

$$((LAMBDA(X)M\{X\})\ Y)$$

and the contractum P' of P is of the form M{Y}, containing j instances
(residuals) $Y_1 \ldots Y_j$ of the argument Y.  By Theorem 5.11 each $Y_i$ may be
contracted in R-* to a normal form $Y_i^*$, such that $A_1^* \gg A_n$ where $A_1^*$ is
the result of replacing each $Y_i$ by $Y_i^*$.  By Corollary 5.10 there exists
an upper bound Y* such that $Y \gg Y^*$ and, for each i, $Y^* \gg Y_i$.  By
contraction of the subexpression Y of $A_0\{Y\}$ we have $A_0\{Y\} \gg A_0\{Y^*\}$; since
Y* is in normal form, the beta contraction of the redex P* in $A_0\{Y^*\}$

$$((LAMBDA(X)M\{X\})\ Y^*)$$

yields a contractum M{Y*} containing j instances of Y*.  But each
instance of Y* may be contracted to the corresponding $Y_i^*$, hence
$A_0\{Y^*\} \gg A_1^*$.  Then we have $A_0\{Y\} \gg A_0\{Y^*\} \gg A_1^* \gg A_n$, and $A_0 \gg A_n$ in R-*.


The simplest illustration of the use of *-conversion to mitigate the beta-R
restriction involves the evaluation of the expression A given by

$$A \equiv ((LAMBDA(X)3)\ B)$$

where

$$B \equiv ((LAMBDA(H)(H\ H))(LAMBDA(H)(H\ H))$$

Since B has no normal form in the conventional lambda calculus (or, as a
consequence, in Either-R) the beta-redex A cannot be contracted under beta-R.
Hence A has no normal form in Either-R; in R-*, however, *-contraction on the
subexpression B of A yields

$$A \nrightarrow ((LAMBDA(X)\ 3)\ *)$$

which may be contracted, under beta-R, to the value 3. We thus can derive the
value 3 from the expression A, despite the restriction on beta conversion. We
may of course derive other normal form values of A which involve the element
*; these may be interpreted as "approximations" of the value of A in the sense
that they retain partial information concerning the value of A. In this light
the expression * itself is a particularly bad approximation of A, as it gives
no clue about the value of A. The expression 3 (which is, significantly,
*-free) is a perfect approximation of A since it contains all of the
information necessary to derive the value of A -- i.e., A=3 in the lambda
calculus.

## 5.1.4:  Consistency of R-* Theories

We observe, at this point, that the addition of the *-conversion axiom to the
lambda calculus does not lead to inconsistency; specifically, if X and Y are
*-free and X→Y in an R-* Theory, then X=Y in the corresponding Lambda
calculus. The intuitive justification for this claim stems from the
unidirectional nature of *-contraction - there is no corresponding abstraction
operation. Thus if the reduction X→Y involves the *-contraction of a
subexpression U, then U must be cancelled since Y is *-free.

The consistency of the R-* Theories follows as a special case of the
consistency of the Either-R-* Theories, which is proved in the next chapter;
consequently no proof is given here.

## 5.2:  Applications to the Lambda Calculus

The theorems of this chapter may provide tools of general usefulness in the study of the conventional lambda calculus. Suppose, for example, that neither of the expressions X and Y have normal forms in the beta-delta calculus, and that furthermore they are not interconvertible. We may still suspect, however, that they are equivalent in an extensional sense. In particular we may wish to prove that if either of $Z\{X\}$ or $Z\{Y\}$ has a normal form in the lambda calculus then $Z\{X\}=Z\{Y\}$.

The mechanism of *-conversion suggests a technique for constructing such proofs. Suppose we could show that in R-* the expressions X and Y have identical sets of normal forms.[1] From Theorem 5.11 it then follows that, for any Z and every $Z^*$ in normal form, $Z\{X\}\gg Z^*$ if and only if $Z\{Y\}\gg Z^*$. But Theorem 5.12 extends this extensional equivalence to the lambda calculus; hence for any Z and any normal form $Z^*$, $Z\{X\}\to Z^*$ if and only if $Z\{Y\}\to Z^*$ where $\to$ denotes lambda calculus reduction. We deduce from these observations that any two expressions which have interconvertible sets of normal forms are eqivalent in this important extensional sense.

We may apply, for sake of illustration, the above technique to the example cited by Scott[2] of the two fixed point operators

$$Y_0 \equiv (LAMBDA(F)(Z\ Z))$$

and

$$Y_1 \equiv (Y_0\ (LAMBDA(Y)(LAMBDA(G)(G\ (Y\ G)))))$$

where Z is the expression

$$(LAMBDA(H)(F\ (H\ H)))$$

$Y_0$ and $Y_1$ are not interconvertible in the lambda calculus, and neither has a normal form. Noting that $Y_0$ contains the single redex (Z Z), the unique single contraction which can be made reduces $Y_0$ to the expression

$$(LAMBDA(F)(F\ (Z\ Z)))$$

---

[1] Specifically, we must show only that $X\to X^*$ implies $Y\to Y^*\to X^*$ and conversely, where $X^*$ and $Y^*$ are any normal form expressions.

[2] Scott[22] credits the example to Corrado Boehm, and acknowledges an unpublished proof due to David Park that the expressions $Y_0$ and $Y_1$ are equivalent in the Scott formalism.

which again contains the single redex (Z Z).  It becomes clear from the
sequence of reductions that this process leads to the conclusion that the
normal forms (in R-*) of $Y_0$ are all of the form

$$(LAMBDA(F)(F\ (F\ (F\ (F\ ...\ (F\ *)\ ...\ )))))$$

and for every natural number n there is a normal form $Y_0*^n$ whose body is F
applied to * n times.

We now refer to the definition of $Y_1$.  By Theorem 5.11, for every normal form
$Y_1'$ of $Y_1\{Y_0\}$ there is a normal form $Y_0*$ such that $Y_1\{Y_0*\}\!\gg\!Y_1'$.  Hence every
normal form of $Y_1$ is a normal form of $Y_1\{Y_0*^n\}$ for some for some n.  But each
of the latter is of the form

$$(G\ (G\ (G\ (G\ ...\ (G\ *)\ ...\ ))))$$

where G stands for the expression (LAMBDA(Y)(LAMBDA(G)(Y G))).  But (G *)
reduces to (LAMBDA(G)(G (* G))) from which, by contraction of its *-redex, we
arrive at $Y_1*1\underset{=}{\equiv}(LAMBDA(G)(G\ *))$.  Then $Y_1*2\underset{=}{\equiv}(G\ Y_1*1)$ has as its maximal normal
form (LAMBDA(G)(G (G *)));  and it becomes clear from this informal argument
that each R-* normal form $Y_1*^n$ of $Y_1$ is of the form

$$(LAMBDA(G)(G\ (G\ (G\ (G\ ...\ (G\ *)\ ...\ )))))$$

whose body contains n applications of G.  Thus each normal form derivable from
$Y_0$ in R-* is derivable from $Y_1$, and conversely.

Now if, for some X, $X\{Y_0\}=X*$ in the lambda calculus where $X*$ is in normal
form, then by Theorem 5.12 $X\{Y_0\}\!\geqslant\!X*$ in R-*.  Then by Theorem 5.11 there is a
normal form $Y_0*^n$ of $Y_0$ such that $X\{Y_0*^n\}\!\gg\!X*$;  since $Y_1$ has a normal form
$Y_1*^m\!\gg\!Y_0*^n$, then $X\{Y_1\}\!\gg\!X*$ hence $X\{Y_1\}=X*$ by the consistency of R-*.  An
entirely symmetric argument shows that $X\{Y_1\}=X*$ implies $X\{Y_0\}=X*$.


## 5.3:  Summary

The mechanism of *-conversion introduced in this chapter allows expressions to
be approximated, to arbitrary accuracy, by expressions in normal form.  The
initial motivation for *-conversion is the mitigation of the limitations on
expressive power imposed by the restricted beta-conversion, but the techniques

of this chapter may be useful generally in the lambda calculus.

The principal technical results of the chapter are:

1) The introduction of * as a canonical representation of the undefined
   (nonterminating) computation, and the axiom on star conversion asserting
   that, for every X, X→*. This axiom is motivated by the interpretation of
   ⇒ as denoting set theoretic inclusion in F*; the empty set, corresponding
   to the undefined computation *, is a subset of every element of F*.

2) Theorem 5.9 and its corollary establish that for any set $A_1$*...$A_n$* of
   normal forms derivable from an expression A in R-*, there exists an
   expression A* in normal form such that A→A* and A*→Ai for each i≤n.

3) Theorem 5.11 shows that if expression X{Y} is reducible to Z*, a normal
   form in R-*, then there exists a normal form Y* such that Y→Y* and
   X{Y*}→Z*. Informally this result assures us that, for every expression Y
   and every context X{Y}, there is a sufficiently good normal form
   approximation Y* of Y. The previous result (2)then guarantees that, for
   any finite set of approximations of Y, we can find a normal form Y*
   which may be used in lieu of any member of the set.

4) Theorem 5.12 provides the final tie to the lambda calculus, by showing
   that every normal form derivable in the lambda calculus is derivable in
   R-*.

The R-* Theory is thus as powerful, in an important sense, as the lambda
calculus with unrestricted beta conversion. Furthermore, the R-* Theories
suggest a natural test for extensional equivalence of expressions: the
interconvertability of normal forms. This technique is applicable to the
lambda calculus, and the extensional equivalence of nonconvertible fixed point
operators $Y_0$ and $Y_1$ is used as an illustration.

The development of *-conversion in Chapter 5 is independent of the EITHER
reduction of the previous chapter. The combination of the two mechanisms is
the project of the next chapter.

## Chapter 6:

## The Either-R-* Theories

The desire for a syntactic basis for a language E, incorporating the EITHER
mechanism informally described in Chapter 3, has led to the presentation (in
Chapter 4) of the Either-R theory. It was noted that the restricted beta
conversion of Either-R limits the usefulness of that theory since, for
example, it prohibits the expression of recursive functions. The inadequacy
of Either-R as a basis for the language E motivated the development, in the
last chapter, of *-conversion. The present chapter brings these efforts to
fruition in the form of the Either-R-* system, which consistently combines
*-conversion with EITHER reduction and provides a satisfactory basis for a
language E.

Specifically, an Either-R-* theory shall consist of the following axioms, each
of which is presented in a previous chapter:

alpha (Ch. 4) interconvertability (by renaming) of congruent expressions --
     e.g. (LAMBDA(X)X) ▪ (LAMBDA(Y)Y);

beta-R (Ch. 4) lambda conversion restricted to redexes whose arguments are
     in normal form -- e.g. ((LAMBDA(X)X) 3)▪3;

various delta axioms (Ch. 4) specifying the interpretation of primitive
     functions and constants -- e.g., (PLUS 3 5) ▪ 8;

epsilon (Ch. 4) contraction of E-redexes-- e.g., (EITHER A B)≯B (Ch. 4);

mu (Ch. 4), abstraction of E-redexes -- e.g. E▪(EITHER E E);

rho (Ch. 4), distribution of function application over terms of an E-redex
     -- e.g. (F (EITHER A B))▪ (EITHER (F A)(F B)).

sigma (Ch. 5) *-contraction -- A≯* for every expresion A.

## 6.1: Consistency of Either-R-*

The consistency of Either-R-* may be established by techniques closely
analogous to the Either-R consistency proof. Recall that the earlier proof
involved the notion of enclosure, and culminated in the implication of
enclosure by ≯ -- i.e., X≯Y in Either-R implies X encloses Y. Extension of

this technique to the present case requires that the mechanism of
*-contraction be accounted for;   accordingly, we extend the notion of
enclosure by


Defn 6.1:   X *-encloses Y if, for each e-residue[1] Y′ of Y, there exists an
      e-residue X′ of X and an expression X* derived from X by *-contraction
      alone, such that X*=Y* in the lambda calculus.


Note that we admit expressions containing the element * in the lambda
calculus, treating * simply as a free variable.   It is clear from the above
definition that *-enclosure is transitive, and that if X encloses Y then X
*-encloses Y.

The following Lemma and its Corollary confirm that *-contraction introduces no
new equivalences in the conventional lambda calculus:


Lemma 6.2:   Let X and Y be *- and EITHER-free expressions, and let X>X* by the
      *-contraction of a subexpression U of X.   If X*=Y in the lambda calculus,
      then X=Y.

   proof: Noting that X* contains a single * (the contractum of U), treating *
      as a variable in the lambda calculus gives us

$$X=((LAMBDA(*)X*)\ U)$$

      by beta conversion.   But X*=Y, hence

$$X=((LAMBDA(*)Y)\ U)$$

      and as Y is *-free the contractum of this beta-redex is simply Y.   Hence
      X=Y.



Corollary 6.3:   If X and Y are *- and EITHER-free and X>X* by a series of
      *-contractions, then X*=Y in the lambda calculus implies X=Y.

   proof is by a simple induction on the number of *-contractions in the

---

[1] Recall Defn 4.7.

reduction sequence from X to X*.

The above lemma and its corollary are hardly counterintuitive in light of the developments of Chapter 5. In particular, it is clear that any occurence of * in X* must be cancelled in the derivation of Y from X, since Y is *-free. Hence we may replace such occurences by arbitrary expressions, which are still cancelled in the derivation of Y; the choice of the homologous subexpressions of X yields X=Y.

The consistency proof for Either-R-* follows the format of the corresponding proof for Either-R, except that the enclosure relation in the latter proof is extended to *-enclosure in the former. The basis of this extension is given by

Lemma 6.4: Let X→Y be a single reduction step in Either-R-*. Then X
       *-encloses Y.

  proof: Lemma 4.13 establishes the lemma for the reductions allowed in
      Either-R; hence we need consider only the case of a *-contraction. Let
      U be the contracted subexpression of X. For each e-residue Y′ of Y,
      there is a corresponding e-residue X′ of X such that either X′ and Y′ are
      identical or Y′ is the result of the *-contraction of an e-residue U′ of
      U in X′. Hence X′→Y′ by *-contraction, and X *-encloses Y.

The following theorem is the Either-R-* analogy of Theorem 4.15:

Thm 6.5: Let X and Y be expressions containing no occurrences of EITHER or *,
      and let X→Y in Either-R-*. Then X=Y in the lambda calculus.

  proof: By Lemma 6.4 and the transitivity of *-enclosure, X *-encloses Y.
      Since each of the expressions X and Y is EITHER-free, each expression is
      its own unique e-residue, and X→X*=Y where X→X* by *-contraction alone.
      By Corollary 6.3, X=Y in the lambda calculus.

Thus the consistency of Either-R-* follows from the consistency of the lambda calculus.

## 6.2:  Relation of * to EITHER

We have already noted that the mechanism of *-contraction leads to the interpretation of each expression A as the upper bound, in the sense of $\geqslant$, of a family of expressions derivable from A.  To formalize the relation between such a family of expressions, we introduce the terminology of

Defn 6.6:   Expressions X and Y are _consistent_ in a theory T if and only if there is an expression Z such that both Z$\geqslant$X and Z$\geqslant$Y in T.

Then the R-* theories are partitioned by the consistency relation into equivalence classes, of which there are infinitely many (since there are infinitely many mutually incongruent normal forms).  Then the characteristic of R-* which is established by Corollary 5.10 is that any finite set of consistent expressions in normal form has an upper bound which is also in normal form.

We note that in R-* the $\geqslant$ ordering on the set of expressions derivable from an expression A is, in general, nontrivial.  Unless A is the element * the upper bound of the set, A, is distinct from the lower bound *; furthermore there may be infinitely many expressions $A_1 \geqslant A_2 \geqslant \ldots$ in the set such that for no j>i is $A_j \geqslant A_i$.  This is certainly not the case in the conventional lambda calculus, in which consistency implies interconvertibility and hence equivalence.  What the mechanism of *-contraction has added to the lambda calculus is a method of deriving from an expression A an approximation A* to A which is strictly weaker in the sense of $\geqslant$.  We may then view the * mechanism as a method of introducing new expressions which are weaker than the conventional lambda calculus expressions, as each expression in R-* is derivable from a *-free expression.

In this light we must regard the EITHER construct as a mechanism for introducing stronger expressions into the lambda calculus.  While R-* (and for that matter the conventional lambda calculus) contain upper bounds only for consistent sets of expressions, we can with EITHER represent the upper bounds of arbitrary (enumerable) sets of expressions.[1]  Observe further that, for

---

[1] Or, equivalently, we may say that in the Either theories, _every_ set of expressions is consistent.

arbitrary expressions X and Y, the expression (EITHER X Y) is the <u>least</u> upper
bound of X and Y since by Theorem 4.4, Z>X and Z>Y implies Z>(EITHER X Y).
This suggests that the ordering of Either-R-* expressions by > forms a
complete lattice.


## 6.3: Evaluators for E

As we have noted, interpreters for languages supporting the EITHER construct
require a slightly different structure from our previous examples: the
reducibility of expressions to multiple values suggests that an evaluator for
E should enumerate the values of the input expression. Accordingly, we
formulate the evaluator as a function E of 2 arguments, an expression X to be
evaluated and a numeric index j specifying which value is to be returned. The
evaluator is constructed such that, for each X and j, E[X;j] is an expression
X′ in normal form such that X>>X′ in Either-R-*. The value of E[X;j] is, in
general, not defined for all values of j;  it may be assumed in particular
that E[X;j] is undefined for those cases of X and j not represented in the
algorithm presented informally below. We again assume the existence of an
invertable pairing function, and use here the notation <n;m> to denote that
natural number which uniquely encodes the ordered pair of natural numbers
(n,m). We make the further assumption that for no n and m is <n;m><2.

      E[X;j] =
           if j=0 then *;
           if X is atomic[1] and j=1 then X;
           if X is of the form (LAMBDA(Y)M) then (LAMBDA(Y)E[M;n]);
           if X is of the form (EITHER A B) and j=<1;n> then E[A;n];
           if X is of the form (EITHER A B) and j=<2;n> then E[B;n];
           if X is of the form (A B) and j =<<m;n>;p> then
                 APPLY[E[A;m];E[B;n];p];

where the algorithm for APPLY is given informally by

      APPLY[F;X;j] =

---

[1] Recall that the atomic expressions are identifiers (including primitive
function symbols and variables) and numeric constants.

```
        if F is of the form (LAMBDA(Y)M) then E[S[X;Y;M];j];
        if (F X) is a delta-redex and j=1 then F[X];
        else if j=1 then (F X);
```

We note that E[X;j] is in normal form where it exists, and the value E[X;j] is in each case the result of an Either-R-* contraction sequence on X. Although we don't claim that the values E[X;j] of X are ordered by > for successively higher values of j, the index j specifies, roughly, which of the approximations of X is to be returned.

We may envision implementations of the E interpreter which make use of massive parallelism to compute simultaneously the values of (F X) for many different approximations of X; such use of redundant computation may serve to minimize the real time required to compute an acceptable value for X. Such an implementation follows, roughly, the spirit of fast adder circuitry which computes redundantly the high order portion of a sum simultaneously with the low order portion, and then selects the correct high order portion on the basis of some intermediate carry. These implementational issues are largely ignored in the present work, but present some intriguing possibilities for future research.

## 6.4: Summary

The Either-R-* Theory may be used as the semantic basis for a language, E, which solves the specific expressibility problem demonstrated in Chapter 4. The evaluation of expressions in E lends itself naturally to the use of multiprocessing techniques which tend to minimize the total real time necessary to relize an acceptable evaluation of an expression (F X) by the simultaneous application of F to one approximation of X while computing a better approximation. While the implementation details are not pursued here, we feel that current technological developments make this area worthy of further study.

## Chapter 7:

## The Either-K Theories

The inconsistency of EITHER distribution (Axiom rho) with the unrestricted beta conversion of the lambda calculus has motivated the restricted beta-R conversion of the systems presented thus far. This chapter explores an alternative formulation, in which EITHER distributivity is sacrificed in order to accommodate the conventional (unrestricted) beta conversion.

The Either-K theories include the axioms alpha, delta, epsilon, mu, and the (unrestricted) beta axiom of the lambda calculi:

Axiom beta: Let E be an expression of the form ((LAMBDA($\underline{a}$)$\underline{b}$)$\underline{c}$). Then E•E´, where E´ is the contractum S[$\underline{c};\underline{a};\underline{b}$].[1]

Since Either-K preserves the axioms of the lambda calculi, it is clear that the equivalence • in Either-K is a proper extension of the lambda calculus equivalence =. In this sense the Either-K calculi are closer to the conventional lambda calculi than the Either-R-* theories.

There is, however, a fundamental sense in which Either-K is a more radical departure from the lambda calculi than is Either-R-*. In the latter theories functions are ultimately applied only to normal form operands whose semantics are those of the lambda calculi. The ability, in Either-K, to apply functions to multivalued expressions (such as E-redexes) requires that we reinterpret the semantics of each function relative to these new elements of its domain.

## 7.1: K-abstraction

By the axiom beta of the lambda calculus, the expressions

M

and

((LAMBDA(x) M) A)

are equivalent when A is an arbitrary expression and M contains no free

---

[1] S is the lambda calculus substitution function given in Defn 2.6.

occurrences of the variable x.  This fact is consistent with the observation
that the bound variable, x, is ignored in the body of the function applied to
A; hence the value of the application is independent of the value of the
argument A.  Despite the intuitive satisfaction with which we accept the above
equivalence, the presence of functions which ignore their arguments
complicates the proof of many otherwise straightforward results in the lambda
calculus.  Indeed, Church has argued against the inclusion of such functions
in his theories, fearing at one time that they led to inconsistencies.[1]

The task of proving the consistency of the Either-K theories, to be attacked
presently, is likewise complicated by the inclusion of functions which ignore
their arguments.  The definitions and results of this section provide the
mechanism for dealing with the formation of such functions in later proofs.
We begin with

Defn 7.1:   A K-redex is an expression of the form

$$((LAMBDA(x)M) \ A)$$

where A is any expression and M is an expression not containing free
occurrences of the variable x.

Defn 7.2:   A K-abstraction is a reduction step[2] consisting of the replacement
of a subexpression M by a K-redex of the form
$$((LAMBDA(x)M) \ A)$$

where A is any expression and x is a variable not occurring free in M.

We now wish to show that the K-abstractions in a reduction sequence can be
postponed to the end of the sequence.  We introduce a term to describe
reduction sequences whose K-abstractions follow all other reductions:

Defn 7.3:   A reduction sequence R is K-normal if no K-abstraction in R

---

[1] For discussion and historical insight, see Curry[12], particularly the
comment at the end of Ch. 3.

[2] recall Defn 4.1.

precedes a reduction step which is not a K-abstraction.

Thus a reduction sequence $X_0 \gg X_1 \gg \ldots \gg X_n$ is K_normal if there is an i, where $0 \leq i \leq n$, such that the reductions $X_0 \gg \ldots X_i$ are not K-abstractions and the reductions $X_i \gg \ldots \gg X_n$ are only K-abstractions. We wish to show that, for every reduction sequence $X_0 \gg \ldots \gg X_n$, there exists a K-normal reduction sequence from $X_0$ to $X_n$. We begin with sequences of length 1:

Thm 7.4:  Let $X_0 \gg X_1 \gg X_2$ be a two-step reduction sequence from $X_0$ to $X_2$, where the reduction step $X_0 \gg X_1$ is a K-abstraction and the reduction step $X_1 \gg X_2$ is not a K-abstraction. Then there is a K-normal reduction sequence from $X_0$ to $X_2$, containing at most one reduction step which is not a K-abstraction.

   proof: Let U be the subexpression of $X_0$ which is replaced in the reduction step $X_0 \gg X_1$. Then U is replaced in this step by U´, an expression of the form

$$((LAMBDA(y)U) \ A)$$

where y is a variable not occurring free in U. We exhaustively examine classes of the reduction step $X_1 \gg X_2$:

Case 1: The reduction step modifies only the subexpression A of U´; let U become A´ in $X_2$. The K-normal sequence from $X_0$ to $X_2$ is then the single K-abstraction replacing U by

$$((LAMBDA(y)U) \ A´)$$

Case 2: The reduction step modifies only the subexpression U of U´; then U becomes W in $X_2$. The K-normal sequence from $X_0$ to $X_2$ is then:

   a) Replace U in $X_0$ by W, yielding $X_0´$;
   b) Replace W in $X_0´$ by the K-redex
$$((LAMBDA(y)W) \ A)$$
      yielding $X_2$.

Case 3: The expression U´ in $X_1$ is replaced by U by beta reduction. Then $X_0$ and $X_2$ are identical expressions, and the empty reduction sequence

yields $X_2$ from $X_0$.

Case 4: The reduction step replaces some subexpression V of $X_1$ by the expression V´, where V is not a subexpression of U´ and U´ is not a subexpression of V. The K-normal sequence from $X_0$ to $X_2$ is then

  a) The replacement of V in $X_0$ by V´, yielding $X_0$´;
  b) The replacement of U in $X_0$´ by U´, yielding $X_2$.

Case 5: The expression U´ is replaced by the expression

$$(EITHER\ U´\ U´)$$

The K-normal sequence from $X_0$ to $X_2$ is then
  a) The replacement of U in $X_0$ by (EITHER U U), yielding $X_0$´;
  b) The replacement of (EITHER U U) in $X_0$´ by (EITHER U´ U´) through two consecutive K-abstractions.

Case 6: The expression U´ is replaced by the expression

$$(EITHER\ ((LAMBDA(y)U)\ A_1)((LAMBDA(y)U)\ A_2)$$

by Axiom rho. The K-normal sequence from $X_0$ to $X_2$ is then
  a) The replacement of U in $X_0$ by (EITHER U U), yielding $X_0$´;
  b) The replacement of (EITHER U U) in $X_0$´ by

$$(EITHER\ ((LAMBDA(y)U)\ A_1)((LAMBDA(y)U)\ A_2))$$

through two consecutive K-abstractions.

Case 7: The subexpression U´ is replaced by an expression W of the form

$$((LAMBDA(z)U)\ A)$$

derived from U´ by alpha conversion. Then the variable z does not occur free in U, and $X_0$ may be reduced to $X_2$ by a single K-abstraction.

Case 8: Some subexpression V containing U´ is replaced by an expression V´. Then one of the following applies:
  8a) V´ is derived from V by alpha conversion. Then we may apply that alpha-conversion to $X_0$, yielding $X_0$´, and follow with the K-abstraction from $X_0$´ to $X_2$.
  8b) V´ contains n occurrences of U´, where n is zero or greater. Then there is a reduction of the same type from $X_0$ to $X_0$´, where $X_0$´ is identical to $X_2$ except for the n occurrences of U in $X_0$´ corresponding to n occurrences of U´ in $X_2$. Our K-normal sequence from $X_0$ to $X_2$

consists of the reduction of $X_0$ to $X_0'$ followed by n K-abstractions replacing the occurrences of U by $U'$.

This list of cases is exhaustive, completing the proof.

Theorem 7.4 shows that every two-step sequence of reductions is equivalent to some K-normal reduction sequence. The generalization of this result to sequences of n reductions is complicated by the fact that the K-normal sequence guaranteed by Theorem 7.4 may be of arbitrary length, thus ruling out a simple induction on the length n of the reduction sequence.

Lemma 7.5:   Let R be a reduction sequence from $X_0$ to $X_n$ containing exactly 1 reduction step which is not a K-abstraction. Then there is a K-normal reduction sequence from $X_0$ to $X_n$.

proof: by induction on the length n of the reduction sequence R.

basis: Trivially true for n<2; for n=2, guaranteed by Theorem 7.4.

induction: Let $X_0 \gg X_1 \gg \ldots \gg X_n$ be the reduction sequence R. If the step $X_0 \gg X_1$ is not a K-abstraction, then R is K-normal; hence we may assume that $X_0 \gg X_1$ is a K-abstraction. Then a single step of the subsequence $X_1 \gg \ldots \gg X_n$ is not a K-abstraction; by the inductive hypothesis, there is a K-normal reduction sequence $X_1 \gg Y_0 \gg Y_1 \gg \ldots \gg X_n$ of which only the reduction step $X_1 \gg Y_0$ may be other than a K-abstraction. But by Theorem 7.4, there is a K-normal sequence $X_0 \gg Z0 \gg \ldots \gg Y_0$ equivalent to the sequence $X_0 \gg X_1 \gg Y_0$; thus the reduction sequence $X_0 \gg Z0 \gg \ldots \gg Y_0 \gg \ldots \gg X_n$ is K-normal from $X_0$ to $X_n$.

Defn 7.6:   The K-index of a reduction sequence R is the number of non-K-abstraction steps in R which follow the first K-abstraction in R. If R contains no K-abstractions, then the K-index of R is zero.

Note that the K-index of a reduction sequence R is zero if and only if R is K-normal. We shall base the induction in the proof of the next theorem on the K-index of the reduction sequence to which it is applied.

Thm 7.7: Let R be a reduction sequence from $X_0$ to $X_n$. Then there is a
K-normal reduction sequence from $X_0$ to $X_n$.

proof is by induction on the K-index of R.

basis: If the K-index of R is zero, then R is K-normal.

induction: The K-index n of R is greater than zero. Let $X_0 \dot{>} \ldots \dot{>} X_n$ denote
R, and let $X_i \dot{>} X_{i+1}$ be the first K-abstraction in R. Let $X_j \dot{>} X_{j+1}$ be the
first reduction step following $X_i \dot{>} X_{i+1}$ in R which is not a K-abstraction;
the existence of such a j is assured by the K-index of R. Then the
subsequence $X_i \dot{>} X_{i+1} \dot{>} \ldots \dot{>} X_j \dot{>} X_{j+1}$ of R contains a single step which is not
a K-abstraction; by Lemma 7.5 there is a K-normal sequence
$X_i \dot{>} Y_0 \dot{>} \ldots \dot{>} X_{j+1}$ from $X_i$ to $X_{j+1}$. Then the sequence R* given by
$X_0 \dot{>} \ldots \dot{>} X_i \dot{>} Y_0 \dot{>} \ldots \dot{>} X_{j+1} \dot{>} \ldots X_n$ has a K-index of n-1. By the induction
hypothesis, there is a K-normal sequence from $X_1$ to $X_n$.

It follows from Theorem 7.7 that every reduction sequence may be reordered in
such a way that every K-abstraction follows every reduction step which is not
a K-abstraction. Curry[12] refers to expressions as fictitious if they appear
as the arguments of K-redexes; hence A is a fictitious subexpression of B if A
is cancelled in the evaluation of B. Theorem 7.7 asserts that the
introduction of fictitious subexpressions can be postponed to the end of a
reduction sequence. Consider the following expressions:

$$Z \equiv (LAMBDA(X)3)$$
$$A \equiv ((LAMBDA(H)(H\ H))(LAMBDA(H)(H\ H)))$$
$$I \equiv (LAMBDA(X)X)$$

Then the reduction sequence

$$3 \dot{>} (Z\ A) \dot{>} (I\ (Z\ A))$$

is not K-normal, since the K-abstraction $3\dot{>}(Z\ A)$ precedes the beta abstraction
$(Z\ A)\dot{>}(I\ (Z\ A))$. We may, however, reorder the sequence so that the fictitious
subexpression A is introduced in the last reduction step; the resulting
reduction sequence

$$3 \dot{>} (I\ 3) \dot{>} (I\ (Z\ A))$$

is K-normal.

## 7.2: Consistency of Either-K Theories

It was noted, following the proof of the consistency of the Either-R theories, that the technique used there was inapplicable to the Either-K axioms since unrestricted beta conversion does not preserve the enclosure relation. We avoid this difficulty in the corresponding proof for the Either-K theories by arranging the reduction sequence of an EITHER-free expression so as to ensure that arguments in beta contractions are unitary. Since the Either-K reduction sequence of an EITHER-free expression can introduce non-unitary subexpressions only through K-abstraction, the result of the preceding section provides a critical step in the present proof.

We begin by distinguishing expressions containing only unitary subexpressions:

Defn 7.8: An expression X is _pure_ if every subexpression of X, including X itself, is unitary.

Note in particular that every EITHER-free expression is pure. We now procede to the major task of this section, which is the proof that the reductions permitted by our axioms preserve purity of expressions. We begin with the case of beta-contractions:

Lemma 7.9: Let Y be EITHER-free and let X be a pure beta-redex of the form

$$((LAMBDA(y)B) \ A)$$

such that for each e-residue $X'$ of X, $X'=Y$. If Z is the result of lambda conversion on X (ie, Z is the result of substituting A for each free y in B), then for every e-residue $Z'$ of Z, $Z'=Y$.

proof: Let $Z'$ be an e-residue of Z. Then $Z'$ contains zero or more occurrences of $A_1$, $A_2$, ..., $A_n$ where each $A_i$ is an e-residue of A. By the purity of X, A is unitary, hence each $A_i$ is convertible to $A_1$. Thus $Z'=Z''$ where $Z''$ is the result of lambda conversion on

$$((LAMBDA(y)B') \ A_1)$$

where $B'$ is some e-residue of B. Hence $Z''=Y$, and $Z'=Y$.

Lemma 7.10:  Let X, Y, Z, and Z´ be as in Lemma 7.9, above.  Then Z is pure.

> proof: Let U be an arbitrary subexpression of Z, and let W be the
> corresponding subexpression of B.  If W contains no occurrences of y
> which are free with respect to X, then W and U are identical, hence U is
> unitary by the purity of X.  If W contains such occurrences of y, then U
> is the result of lambda conversion on
>
> $$((LAMBDA(y)W) A)$$
>
> and, by Lemma 7.9, U is unitary.

We next show that beta abstractions preserve purity, so long as they are not
K-abstractions:

Lemma 7.11:  Let Z be a pure expression containing 1 or more occurrences of
the subexpression A.  Let W be a beta-redex of the form

$$((LAMBDA(Y)B) A)$$

such that the contractum of W is Z.  Then W is pure and, for every
e-residue W´ of W there exists an e-residue Z´ of Z such that W´=Z´.

> proof: Since A is a subexpression of the pure expression Z, A is unitary;
> let the e-residues $A_1´$, $A_2´$,...$A_k´$ of A each be convertible to A´ in the
> lambda calculus.  For each e-residue B´ of B there is a corresponding
> e-residue Z´ of Z, such that Z´ contains some $A_j´$ in place of each free
> occurrence of Y in B;  hence Z´=S[A´;y;B´].  Each e-residue W´ is of the
> form $((LA_1´MBDA_1´(Y)B´)A_1´)$ where B´ is an e-residue of B; but then W´ is
> convertible to S[A´;y;B´]=Z´.  Thus each e-residue W´ of W is convertible
> to an e-residue Z´ of Z.  Noting that homologous subexpressions B1 and Z1
> of B and Z, respectively, are either identical or related by
> Z1=S[A´;Y;B1], we deduce by the above argument and the purity of Z that B
> is pure.  Hence W is pure.

Note that Lemma 7.11 fails to hold for K-abstractions;  consider, for example,
the K-abstraction

$$M \rightarrow ((\text{LAMBDA}(X)M)(\text{EITHER } 2 \ 3))$$

where M contains no free occurrences of the variable X. Clearly the
abstraction of M is impure regardless of the purity of M. We now present the
principal result of this section, from which the consistency of the Either-R
axioms follows directly:

Lemma 7.12: Let $X \rightarrow Y$ be a single reduction step other than a K-abstraction in
Either-K, and let X be pure. Then Y is pure and X encloses Y.

  proof: The cases where $X \rightarrow Y$ is a beta conversion follow directly from Lemmas
    7.9, 7.10, and 7.11; and if the step is an alpha conversion, the
    e-residues of Y are clearly congruent to the e-residues of X, and Y is
    pure. If $X \rightarrow Y$ is a delta conversion then both X and Y are EITHER-free and
    the lemma is trivially true. If $X \rightarrow Y$ is an EITHER-conversion in either
    direction, the purity of Y follows from the purity of X and the
    e-residues of X and Y are identical.

The consistency of the Either-K theories is presented as

Thm 7.13: Let X and Y be EITHER-free expressions, and let $X \rightarrow Y$ in Either-K.
    Then X=Y in the lambda calculus.

  proof: From Theorem 7.7, we may assume that there is a K-normal reduction
    sequence from X to Y; let $X \rightarrow \ldots \rightarrow X_1 \rightarrow Y_0 \rightarrow \ldots \rightarrow Y$ be such a sequence, where the
    subsequence $X \rightarrow \ldots \rightarrow Y_0$ contains no K-abstractions and $Y_0 \rightarrow \ldots \rightarrow Y$ contains
    only K-abstractions. Then $Y_0$ must be EITHER-free, since each of the
    K-abstractions $Y_i \rightarrow Y_{i+1}$ can only increase the number of EITHER redexes,
    and Y is EITHER-free. $Y_0 = Y$ in the lambda calculus since each of the
    conversions $Y_0 \rightarrow \ldots \rightarrow Y$ is a valid beta conversion. By Lemma 7.12, X must
    enclose $Y_0$ since X is pure; but each of these expressions is EITHER-free
    and hence is its own e-residue. Thus $X = Y_0 = Y$.

Corollary 7.14: Let X and Y be EITHER-free expressions, and let $X \bullet Y$ in
    Either-K. Then X=Y in the lambda calculus.

proof: Directly from Corollary 7.13.


## 7.3: Functional Domains of Either-K

The semantics of the Either-K Theories bear a superficial similarity to those
of the corresponding Either-R-* Theories: in each case a functional domain F
of the lambda calculus is extended to a domain F* whose elements are
enumerable subsets of F. The question of restrictions on beta conversion
seems, at first glance, to be an issue of evaluation order whose semantic
ramifications parallel, say, those of the applicative/normal order
distinction. While this analogy can be defended, as it has been in earlier
sections of this thesis, there is evidence suggesting that the distinction
between the Either-R and Either-K semantics is of a rather more fundamental
nature.

The distributivity of function application over EITHER terms, sanctioned in
the Either-R Theories by Axiom rho, constitutes a limitation on the expressive
power of languages built on these theories. Consider, for example, the
function f whose informal definition is

$$f[x] = x+x;$$

which computes, in the lambda calculus, a numeric value which is twice the
value of its argument x. Our experience with conventional applicative
languages reinforces an intuitive expectation that f will have only even
numbers in its range (assuming that the domain of f is the set of natural
numbers). The natural extension of our intuition to the Either-R Theories is
consistent with the range of f there, containing enumerable sets of even
numbers. In the Either-K Theories, however, we must realign our intuition.
The application of f to the argument either[2;3], for example, is reducible in
Either-K to any of the numbers in {4,5,6} rather than the {4,6} result of
Either-R. Thus although the semantics of the application of functions to
single-valued arguments remains consistent with the lambda calculus, the
behavior of functions with multivalued arguments differs between the Either-R
and Either-K systems.

A more bizarre demonstration of this difference is the function $g$ defined informally by

$$g[x] = \quad \text{if } x > x \text{ then } 1;$$
$$\text{else } 0;$$

which, in the lambda and Either-R calculi is equivalent to the single argument constant function which always returns zero. Yet the Either-K reduction of $g[either[1;2]]$ yields the values $\{0,1\}$, even though $g[1]$ and $g[2]$ each evaluate to $\{0\}$. Since the behavior of $g$ in Either-K violates the distributivity axiom of the Either-R Theories, we clearly cannot express in these theories a function with the properties of $g$; yet $g$ appears to be a computable function definable on the domain $F^*$.


## 7.4:  Summary

This chapter presents a consistent theory which combines EITHER conversion with unrestricted beta conversion. This combination requires 1) that we abandon the distributivity of functions over EITHER terms, and 2) that we reinterpret the semantics of EITHER. The latter reinterpretation is only hinted at in this chapter, and we confess that the semantics of the Either-K theories require further study.

*This empty page was substituted for a blank page in the original document.*

## Chapter 8:
## Summary and Conclusions

There has been a definite tendency, in the course of the work reported here, to provide questions much more frequently than answers. We regard this situation, perhaps defensively, as a healthy attribute of research in a field as theoretically immature as the science of programming languages.


### 8.1: Summary

The general topic of this thesis is the correspondence between the syntactic mechanism of an interpreter and the semantic structure of the language it interprets. The restriction of this study to the class of applicative languages is defended, in Chapter 1, on the grounds that

i) Interpretive mechanism for applicative languages is simple, since such complications as assignment, side effects, and transfers of control are avoided;

ii) The semantics of applicative languages are independent of the notion of time;

iii) The theories of mathematical functions may serve as a semantic basis for applicative languages.

Expressions of an applicative language are viewed as representations of objects in an abstract semantic _functional domain_ containing functions and constants, and expressions are semantically equivalent if they represent the same abstract element.

The stack- and tree-environment interpreters presented in Chapter 2 illustrate semantic limitations imposed by typical compromises between efficiency and expressive power. The defect of $S^1$ must be viewed as an interpreter "bug" if we take mathematical functions as a semantic basis, since certain expressions are interpreted by S in a manner inconsistent with the behavior of functions.

The T interpreter of Chapter 2 relates the issue of evaluation order to the expressibility of certain functions. The applicative order evaluation of T,

---

[1] i.e., the FUNARG problem.

in which arguments to a function are evaluated before the application of the
function, is seen to lead to the inexpressibility of functions which ignore
the value of their arguments.  This motivates a preference for the normal
order evaluation of the N model, in which such functions are expressible.  The
demonstration in chapter 2 of a functional domain F of N assures us that every
expression is interpreted by N in a way that is consistent with our functional
semantics; it does not, however, establish that every valid semantic element
(e.g., every computable function defined on the semantic domain of N) is
expressible in N.

Chapter 3 demonstrates a function, WHICHFF, which despite its computability is
expressible neither in N nor in the lambda calculus.  The expressibility of
WHICHFF seems to require a mechanism analogous to multiprocessing, and two
therapeutic language extensions are considered:

   i) A "coding" primitive which allows a program access to the _representation_
      of a function supplied as its argument; and

   ii) A primitive EITHER whose interpretation involves the dovetailed
      evaluation of its arguments.

The admission of coding essentially abandons all semantic constraints and
allows the programmer to reinterpret expressions as he wishes; we thus discard
this alternative as semantic anarchy.  The EITHER primitive may be justified
in terms of applicative semantics, however, by the expansion of the semantic
domain F into the power set F*, each of whose elements is a subset of F.  Thus
once EITHER is introduced we must semantically associate each expression X
with an enumerable set of abstract values or "meanings" of X.  Such a
multivalued semantic domain is necessary to reconcile the function WHICHFF
with applicative language semantics.

The semantic domain F* motivated in Chapter 3 is suggestive of a complete
lattice ordered by set theoretic inclusion.  The undefined (or nonterminating)
computation is naturally associated with the empty set in F*, and that
expression TOP whose values include the entire domain of the lambda calculus
corresponds to the maximal element of F*.  The semantic element associated
with the expression either[a;b] becomes the union of the respective F*
elements corresponding to the expressions a and b.

In Chapter 4 our attention returns to the subject of interpretive mechanisms.
In particular we desire a formalism for syntactic manipulation of expressions
in a language including EITHER, reflecting the insight gained through informal
scrutiny of the structure of F* in Chapter 3. The formalisms introduced in
Chapters 4-7 are systems of conversion axioms, similar to (and based on) the
lambda calculus; each system (or theory) defines an ordering, $\geqslant$, corresponding
to inclusion in F* -- thus, for example, either[a;b]$\geqslant$a and either[a;b]$\geqslant$b in
each system.

A complication arising in Chapter 4 involves the reconciliation of the beta
reduction[1] of the lambda calculus with the intuitively motivated requirement
that functions be distributive over EITHER terms -- i.e., that f[either[a;b]]
be equivalent to either[f[a];f[b]]. The EITHER-R system presented in Chapter
4 resolves this difficulty by restricting beta conversion to arguments which
are reduced to normal form; while consistent, the resulting theory is too weak
to be useful.

The syntactic mechanism of *-conversion, presented in Chapter 5, solves this
problem of Either-R. Chapter 5 introduces the expression * as a canonical
(normal form) representation of the undefined computation, and extends the
ordering $\geqslant$ so that the syntactic significance of * (A$\geqslant$* for every expression
A) reflects the semantic significance of the undefined computation (the empty
set is a subset of every element of F*). The use of *-reduction allows every
expression, including the single-valued expressions of the conventional lambda
calculus, to be reduced to multiple normal forms. The R-* theory developed in
Chapter 5 reinforces an interpretation of the normal forms derivable from an
expression X as approximations to X, and shows that for any context A{X}
having normal form value A´ there exists a sufficiently good (normal form)
approximation X* of X such that A{X*} also has the value A´. This result has
major semantic consequences; in particular, it implies that meaning of an
expression X is completely characterized by the set of normal forms derivable
(in R-*) from X. Moreover the result is shown to carry over to the
conventional lambda calculus, since every normal form derivable in the lambda
calculus is derivable in R-*. The extensional semantic equivalence relation

---

[1] Informally, beta reduction is the application of a lambda expression
(user-defined function) by substitution of its argument for free occurences of
the bound variable in the body of the lambda expression.

suggested by these findings, namely the interconvertability of normal forms derivable in R-*, is demonstrated by showing the equivalence of non-interconvertable expressions for the fixed point operator Y.

The mechanisms of *-conversion and EITHER-reduction are combined, in Chapter 6, to yield the Either-R-* system. The respective functions of the two mechanisms are, in a sense, complementary; roughly speaking EITHER allows expressions to be combined to make "stronger" expressions while *-conversion allows expressions to be resolved into weaker component expressions. The Either-R-* system is consistent, retains the power of the lambda calculus, and interprets EITHER according to the semantic notions of Chapter 3. We thus view Either-R-* as a practical syntactic basis for the construction of for interpreters of languages based on multivalued semantic domains; such an interpreter, E, is presented at the end of Chapter 6.

Chapter 7 explores an alternative resolution of the conflict between unrestricted beta conversion and the distributivity of functions over EITHER terms. The Either-K system presented in that chapter sacrifices such distributivity in order to allow the unrestricted beta conversion of the lambda calculus. While this combination results in a consistent theory (as demonstrated in Chapter 7) it leads to a semantic structure which is fundamentally different from that of the Either-R theories, in particular regarding the application of functions to multivalued arguments.


8.2: Conclusions

The study of applicative languages from the complementary viewpoints of interpretive and semantic structure leads synergistically, we feel, to a new insight in each area. We have repeatedly found the syntactic mechanisms and semantic structures to be mutually illuminating, and view this dual perspective as a principal influence on the direction and motivation of this thesis.

The following are viewed as the principal results of this thesis:

1) The motivation and presentation of an applicative model of multiprocessing. The applicative approach to this mechanism has certain

technical advantages over conventional formulations; notable among these
is the complete irrelevance of time as a parameter of language semantics.
The corollary disadvantage of the applicative model is its uselessness in
the study of time dependent implementation considerations -- such as
scheduling, deadlocks, and synchrony of processes.

2) The formulation of the semantic domain F* for multivalued applicative
   languages. We find particularly interesting the potential extension of
   the Scott formalism which F* suggests: we have added, to the Scott
   domain, unique upper bounds of arbitrary sets of semantically distinct
   elements. The lack of such upper bounds in the Scott model has been
   conspicuous, and the EITHER construct presented here seems to provide a
   natural interpretation for them.                              .

3) The mechanism of *-conversion and the results relating it to the
   conventional lambda calculus. These results augment the lambda calculus
   with a syntactic substructure (i.e., the ordering under $\geqslant$) which bears
   close analogy to the semantic structure developed by Scott. In addition,
   *-conversion provides a concrete (syntactic) relation of semantic
   equivalence which may illuminate the relationship between lambda calculus
   expressions having no normal forms.

4) The presentation of consistent theories of EITHER conversion. The
   analyses of these systems is by no means exhaustive; we have not shown,
   for example, that no axiom is derivable from the remaining axioms. The
   theories do, however, provide sufficiently powerful syntactic mechanism
   that interpreters may realistically be based upon them.


## 8.3: Directions of Future Research

We recognize that this section constitutes fertile grounds for an essay strewn
with universal quantifiers. Restricting our attention to specific questions
left unanswered by this work, we find most demanding of further attention:

1) The relative expressive power of EITHER-augmented versus CODE-augmented
   languages. We conjecture that every computable function defined on the
   single-valued domain of the lambda calculus is expressible in the
   language E, and have in fact spent considerable effort in trying

(unsuccessfully) to prove this conjecture. The discovery of computable functions expressible (with coding) in C but inexpressible (with EITHER) in E would be counterintuitive and somewhat depressing.

2) The semantics and expressive power of languages based on the Either-K Theories. The presence of functions which compute different results for a multivalued argument X than for singlevalued components of X raises new fundamental questions: what is a computable function on $F^*$? Are the Either-K Theories functionally complete? If not (and we are pessimistic on that issue) which functions are not expressible in Either-K?

3) There appears to be a great deal of room for further development of the theories of EITHER conversion. The extension of these theories to allow eta reduction seems feasible. Further extensions may make the extensional relation of semantic equivalence tractable by syntactic means alone, e.g. by axiomatically asserting in Either-R-$^*$ the equivalence of expressions whose normal forms are interconvertable.

4) The area of interpretive mechanisms for EITHER-based languages has some interesting possibilities. The techniques of computational complexity studies, for example, might yield some quantitative bounds on the computation time necessary for the evaluation of classes of applicative expressions. As the cost of computation power continues to plummet, methods for making use of massive parallelism becomes a practical as well as academic interest.

5) The relationship between the mechanisms of EITHER- and $^*$-conversion and the semantic constructions of Scott demand more serious attention than the informal parallels drawn here. Much of Scott's important work seems to bear rather directly on the systems presented here, and we recognize that too little advantage has been taken of this resource.

It must finally be acknowledged that our quest for a functionally complete language -- one whose domain D contains every computable function defined on D -- has not been an unqualified success. The lambda calculus, whose functional completeness was suspect, was scrutinized and found to be incapable of expressing certain functions (e.g. WHICHFF). To remedy this inadequacy, the lambda calculus was extended via the EITHER construct; the result (the Either

theories) is, indeed, capable of expressing WHICHFF. However, the new systems have additional elements in their domain, so that the functional completeness of the Either theories is again suspect. The results of this thesis, then, suggest a similar program of scrutiny and extension to repair their inadequacies. There is an inevitable circularity in this course of research, mitigated by the fact that each cycle allows us to see previous cycles more clearly.

> A way a lone a last a loved along the/
> riverrun, past Eve's and Adam's, from
> swerve of shore to bend of bay, brings
> us by a commodius vicus of recirculation
> back to Howthe Castle and Environs.

> -Finnegan's Wake,
> last/first lines

References

[1] Church, A., _The Calculi of Lambda Conversion_, Annals of Mathematics
      Studies, Princeton University Press 1941.

[2] Landin, P, "A lambda-calculus Approach" in _Advances in Programming and
      Non-numerical Computation_, Permagon Press, New York 1966.

[3] Dertouzos, M., _Structure and Interpretation of Computer Languages_ (class
      notes for M.I.T. subject 6.252) Spring, 1973.

[4] Perlis, A. J., "The Synthesis of Algorithmic Systems", JACM, January 1967.

[5] Scott, D., _Outline of a Mathematical Theory of Computation_, Technical
      monograph PRG-2, Oxford University Nov 1970.

[6] Scott, D., _The Lattice of Flow Diagrams_, Technical monograph PRG-3, Oxford
      University November 1970.

[7] Hoare, C. A. R., "Procedures and Parameters: An Axiomatic Approach",
      _Lecture Notes in Mathematics_ 188, Springer-Verlag, Berlin 1971.

[8] Hewitt, C., _Description and Theoretical Analysis (using Schemata) of
      PLANNER: A Language for Proving Theorems and Manipulating Models in
      a Robot_, M.I.T. Artificial Intelligence Laboratory TR-258, April
      1972.

[9] Cooper, D. C., "Program Schemes, Programs and Logic", _Lecture Notes in
      Mathematics_ 188, Springer-Verlag, Berlin, 1971.

[10] Moses, J., "The Function of FUNCTION in LISP (or, Why the FUNARG Problem
      Should Be Called the Environment Problem)", SIGSAM Bulletin 15, July
      1970.

[11] Rosenbloom, P. C., _The Elements of Mathematical Logic_, Dover
      Publications, New York 1950.

[12] Curry, H. B., and Feys, R., _Combinatory Logic_, Amsterdam, 1958.

[13] Paterson, M.S., "Program Schemata", _Machine Intelligence III_, Edinburgh
      University Press 1968.

[14] Strachey, C., "Fundamental Concepts in Programming Languages", NATO
      Conference, Copenhagen, 1967.

[15] Morris, J., _Lambda-Calculus Models of Programming Languages_, PhD Thesis,
      M.I.T. December 1968.

[16] Schwartz, J.T., "Semantic Definition Methods and the Evolution of
      Programming Languages" in _Formal Semantics of Programming Languages_,
      Prentice-Hall 1972.

[17] Ershov, A.P., "Theory of Program Schemata", IFIP Congress 71, August
      1971.

[18] Landin, P.J., "The Next 700 Programming Languages", CACM March 1966.

[19] Wegner, P., "Programming Language Semantics", in _Formal Semantics of
      Programming Languages_, Prentice-Hall 1972.

[20] Boehm, C. "Alcune Proprieta Delle Forme beta-eta-normali nel
      lambda-K-calculo," Consiglio nazionale delle ricera Roma 696, 1968.

[21] Hindley, J.R. et al, _Introduction to Combinatory Logic_, Cambridge
      University Press 1972.

[22] Scott, D. "Lattice Theory, Data Types and Semantics" in _Formal Semantics
      of Programming Languages_, Prentice-Hall 1972.

[23] Weizenbaum, J. "The FUNARG Problem Explained", unpublished memorandum, MIT 1968.

[24] Walk, K. et al, "Abstract Syntax and Interpretation of PL/1, Version III," IBM Laboratory, Vienna TR25.098, 1969.

[25] Floyd, R.W. "Assigning Meanings to Programs," Proc. Symposium on Appl. Math. volume 19, 1967.

[26] McCarthy, J. et al, The LISP 1.5 Programming Manual MIT Press 1965.

[27] Curry, H.B. et al, Combinatory Logic vol. II, Amsterdam, 1972.