LAMBDA-CALCULUS MODELS OF PROGRAMMING LANGUAGES

by

JAMES HIRAM MORRIS, JR.

B.S., Carnegie Institute of Technology
(1963)

S.M., Massachusetts Institute of Technology
(1966)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December, 1968, *i.e. Feb. 1969*

Signature of Author _____
Alfre .968

Certified by _____
.sor

Accepted by _____
Chairman, Departmental Committee on Graduate Studies

LAMBDA-CALCULUS MODELS OF PROGRAMMING LANGUAGES

by

JAMES HIRAM MORRIS, JR.

Submitted to the Alfred P. Sloan School of Management on December 13, 1968 in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

A B S T R A C T

Two aspect of programming languages, recursive definitions and type declarations are analyzed in detail. Church's λ-calculus is used as a model of a programming language for purposes of the analysis.

The main result on recursion is an analogue to Kleene's first recursion theorem: If A = FA for any λ-expressions A and F, then A is an extension of YF in the sense that if E[YF], any expression containing YF, has a normal form then E[YF] = E[A]. Y is Curry's paradoxical combinator. The result is shown to be invariant for many different versions of Y.

A system of types and type declarations is developed for the λ-calculus and its semantic assumptions are identified. The system is shown to be adequate in the sense that it permits a preprocessor to check formulae prior to evaluation to prevent type errors. It is shown that any formula with a valid assignment of types to all its subexpressions must have a normal form.

Thesis Supervisor: John M. Wozencraft
Title: Professor of Electrical Engineering

ACKNOWLEDGEMENT

I express my appreciation to Professor John M. Wozencraft for his active interest in and unfailing support for my thesis research. Professors Arthur Evans, Jr. and Robert M. Graham have contributed significantly to the successful completion of this dissertation.

A special acknowledgement is due Peter J. Landin, whose incisive analyses of programming languages have been an inspiration for my work.

I am grateful to Project MAC for financial support and for the stimulating environment it has afforded me.

Finally, I shall be forever indebted to my wife, Susan. Not only has she been a constant source of encouragement and understanding, but she must have typed over a thousand λ-expressions in the past year!

Work reported herein was supported in part by Project MAC, an M.I.T research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01). Reproduction of this report, in whole or in part, is permitted for any purpose of the United States Government.

C O N T E N T S

ABSTRACT 2

ACKNOWLEDGMENTS 3

I. INTRODUCTION

Synthesis and Analysis of Programming Languages 6
Semantics and Pragmatics 9
Thesis 12

II. THE LAMBDA-CALCULUS

Expressions as Formal Objects 13
Conversion Rules 18
Basic Theorems 24
Lambda Expressions Gödelized 31
Lambda-Calculus as a Model of a Programming Language 37

III. RECURSION

Two Views of Recursion 40
Extensional Equivalence 49
The Minimality of Fixed-Points Produced by Y 54
Conclusions 76

IV. TYPES

Motivation 79
Perspective 81
The Basic System 86
Sufficiency of the Type System 96
An Algorithm for Type Assignments 100
Drawbacks of The Type System 106
An Extended System 112
Summary 118

V. CONCLUSIONS

Support for the Thesis	119
Directions for Future Research	121
Remarks	127
REFERENCES	128
BIOGRAPHICAL NOTE	130

CHAPTER 1

Introduction

This dissertation presents a study of two well-known features of programming languages, recursion and types. The main tool in this work is Church's λ-calculus [1] which we use as the model of a programming language.

A. Synthesis and Analysis of Programming Languages.

Our work is almost entirely analytic. We do not invent a new language or add features to an existing one; but, rather, explore in depth the two mechanisms mentioned above.

Since programming languages do not appear spontaneously, as do natural languages, the language analyzer is somewhat dependent upon the synthesizer for source material. In the somewhat acerbic words of A. J. Perlis (a master-synthesizer), "In order that their [the analyzers'] research continue to progress, it is necessary that we ... operate very rapidly in building bigger and better languages. If we do not, I am very much afraid that they will run out of abstractions to play with." [2]

At present, however, there seem to be more than enough programming languages to satisfy the needs of analyzers and programmers

alike. Indeed, the would-be computer user is offered a perplexing array of languages to use. Clearly, there is need for an identification of the principles (if any) behind programming languages and their various mechanisms. The aim of such work need not be the development of the long sought universal programming language. It is unlikely that such a language is either possible or desirable.

A reasonable goal is simply the unification and rationalization of the many facets of mechanical languages. The beneficiaries of these efforts include not only language users, but also the designers themselves. A consistent set of principles can aid the design, implementation and documentation of a language by resolving many of the design decisions in a consistent way. Which set of principles one adopts does not seem to be as critical as remaining faithful to the ones chosen. A consistent policy of design guards the user from "surprises" after he has grasped the essence of a language. Consider, for example, the many surprises awaiting an ALGOL user who understands functions but has not learned about side effects.

One of the most fruitful techniques of language analysis is explication through elimination. The basic idea is that one explains a linguistic feature by showing how one could do without it. For example, one explains a FORTRAN DO-loop by showing how it can be simulated by IF-statements, transfers, and assignments. The important point to remember about such explications is that they do not purport to define the notion in question but only to explain it. For example, if the DO-loop

were construed as merely an abbreviation for a certain configuration of IF-statements, transfers and assignments then one would expect that transfers into the scope of a DO-loop were allowed (which they are not). Similarly, if one took von Neumann's explication of the ordered pair as its definition, he would expect that the set union of two ordered pairs was meaningful.+ (See Quine [3, page 257] for an illuminating discussion of this issue.)

The products of analysis by reduction to simpler mechanisms can be fed back into the design process in the following way: Once we have reduced the language in question to a simpler one, we can build up a new language by defining new facilities in terms of the primitive base. This new language is likely to have the same flavor as the old one but may include genuinely original features suggested by the primitive basis.

This phenomenon manifests itself in Landin's development of ISWIM [4 , 5]. Fast on the heels of his explication of ALGOL-60 in terms of the λ-calculus came his presentation of ISWIM, an interesting language whose various mechanisms appear to mesh nicely.

In Chapter III we shall consider the mechanism of recursion and its simulation in the λ-calculus. While this explication should not be taken as <u>the</u> definition of recursion, it offers interesting possibilities for extensions of the notion. For example, it allows one to make some sense of the following, rather suspicious, recursive definition:

$$f(x) = \underline{\text{if}}\ x=0\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ f.$$

+ The pair $\langle a,b\rangle$ can be thought of as the set of sets $\{\{a\}, \{a,b\}\}$ since this representation fulfills the basic requirement that $\langle a,b\rangle = \langle c,d\rangle$ if and only if $a=c$ and $b=d$.

B. Semantics and Pragmatics

It is commonly accepted that the semantics of a programming language have to do with the meaning of the legal programs. In practice it is a catch-all term with which we designate all the aspects of the language other than the obviously syntactical. We raise a quibble with this usage in order to emphasize our point of view.

We interpret "semantics" in the narrow sense of C. W. Morris [6] and define our terms as follows:

1. Syntax delineates the legal forms, or utterances, of a language.
2. Semantics treats with the relation between the forms, or expressions, in the language and the objects they denote.
3. Pragmatics treats with the relation between a language and its users (human or mechanical).

We argue that from a strictly operational point of view, semantics are unnecessary. To specify a language, we need only give its syntax and pragmatics. The pragmatics are most easily specified by how a machine, real or contrived, will react when presented with a legal form of the language. This provides all of the information that a user or implementor needs to know about the language.

The objects or values that certain expressions may denote are entirely in the mind of the human interpreter of the language. It would be futile to examine an on-going machine computation in search of these objects. The machine is simply reading and writing symbols.

If anything is manipulating these illusive objects, it must be Koestler's "ghost in the machine" [7].

Despite the fact that semantics (in the narrow sense above) appear superfluous to the specification and operation of a programming language, we believe that they are important. Specifically, we hold that the linguistic forms of a programming language should be construed as denoting objects whenever possible and that the pragmatics (i.e., the treatment of expressions by machines) should be consistent with the semantics.

The practical basis for this position is a simple one; people find it convenient to think in terms of objects and operations on these objects, and a programming language designed for people should reflect that predeliction.+

As evidence for this assertion let us consider a few programming languages with respect to their semantics and their acceptance by computer users.

It seems fair to say that the average computer user prefers FORTRAN to machine code. We see the major distinction between these alternatives as FORTRAN's use of algebraic expressions; the mechanisms for branching and transfer of control are roughly equivalent. The important thing is that they denote things - numbers - while a sequence of computer instructions does not. The fact that FORTRAN's notation is traditional we consider less vital; we would be almost as

---

+ Whorf [8] has presented evidence that certain non-Indo-European languages (and hence their users) make no commitment to the existence of objects. If this be the case, we happily restrict our considerations to programming languages tailored to Indo-European computer users.

happy with "sum(x,y)" as "x+y" since either one denotes an object.

LISP[9] seems to have gained wider acceptance than IPL-V[10]. Both languages offer approximately the same pragmatics (list processing) but LISP has semantics while IPL-V does not. IPL-V is modeled after an assembly language for a computer which operates on list-structures, and there are few forms in the language which have denotations. LISP is an expression evaluator, and its expressions denote list structures. We attribute part of the relative success of LISP to this distinction rather than to its use of more mnemonic identifiers or other improvements over IPL-V.

C. The Thesis

We view the work presented in this dissertation as a semantic analysis of the two subjects, recursion and data types. We tend to understand these subjects pragmatically. When a programmer thinks of recursion, he thinks of push-down stacks and other aspects of how recursion "works". Similarly, types and type declarations are often described as communications to a compiler to aid it in allocating storage, etc.

The thesis of this dissertation, then, is that these aspects of programming languages can be given an intuitively reasonable semantic interpretation.

Specifically, in Chapter 3 we show how a function defined by recursion can be viewed as the result of a particular operation on an object called a functional. In Chapter 4 we outline a type declaration system in which a type can be construed as a set of objects and a type declaration as an assertion that a particular expression always denotes a member of the set.

The analysis is not performed on a real programming language but on the λ-calculus. Chapter 2 presents background on the λ-calculus and how it relates to programming languages. The simplicity of the λ-calculus makes it more amenable to rigorous treatment than even a well-defined programming language (e.g. ALGOL). Of the many formalisms used in meta-mathematics it appears to bear the strongest resemblance to a programming language because the notions of variables and functional application are quite explicit in its formulation.

CHAPTER II

The λ-Calculus

In this chapter we summarize a number of facts about λ-calculi which will be referred to in later chapters. The treatment here is necessarily abbreviated, and the interested reader is referred to [1] or [11] where thorough treatments of the subject are given.

A. Expressions as Formal Objects

Since most of the work in this thesis is involved with manipulation of expressions, it is advisable to define the notions of expressions, subexpressions, etc. in a precise way. To illustrate the concepts introduced, we shall present the definition of a class of expressions called well-formed expressions (wfes) which constitute the object language of a λ-calculus. Although this single class is the only one we shall deal with extensively, it seems reasonable to present the supporting concepts as general notions.

1. Operators. A class of expressions is built up from a (possibly infinite) set of tokens called operators. Each operator has a fixed, finite degree ≥ 0 which indicates how many constituent expressions it "takes". An operator of degree 0 is called an atom. A formal expression is either an atom or an operator of degree n, together with n expressions called its constituents. Certain restrictions may

be placed on a class of formal expressions to single out sub-classes of interest.

The operators for wfes consist of

(a) λ which has degree 2

(b) γ which has degree 2

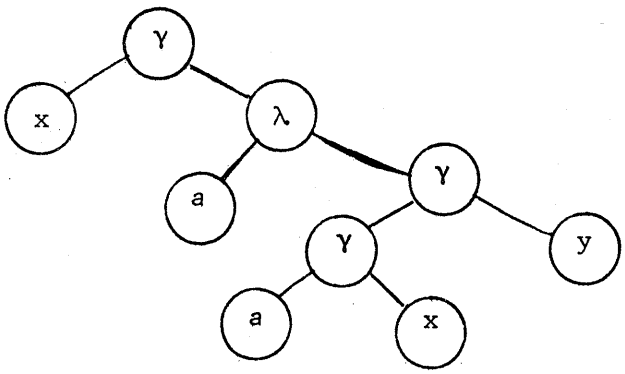
(c) an infinite number of atoms called variables:

a,b,...,z,a',b',...

(d) an infinite number of atoms called constants. The tokens for constants may vary and will be specified in any context where constants are relevant. For example, 0, 1, 2, ... might be constants.

The first constituent of a λ-expression must be a variable.

A specimen of an expression may be presented in a variety of ways. We shall employ two representations, either trees or linear symbol strings. For example, the following are two presentations of the same wfe:



$x(\lambda a.axy)$

The rules governing tree representations of wfe are simple. Each node contains an operator and has a number of depending branches equal to the degree of the operator. The sub-trees below a given node are its constituents and are assumed to have a fixed order, given by their left to right ordering on the page.

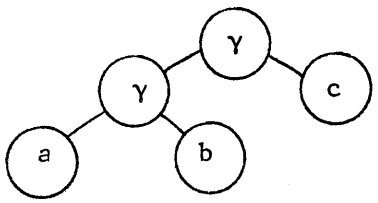
To establish the rules for the linear representation, we present the following BNF definition of wfe.

< wfe > ::= λ < variable > . < wfe > | < combination >

< combination > ::= < combination > < atom > | < atom >

< atom > ::= < variable > | < constant > | (< wfe >)

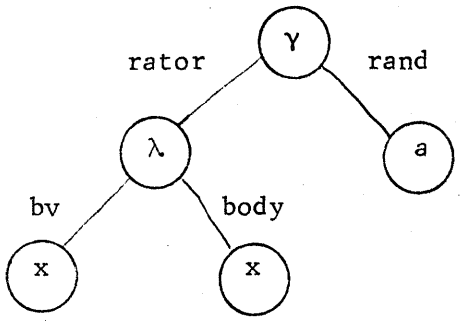
Notice that combinations nest to the left, i.e., a b c and (a b) c denote the same wfe and correspond to the same tree:



2. Branches and Paths. As part of the meta-language with which we discuss expressions, we introduce a set of tokens called branch names. These are used to single out particular components of expressions.

The branch names for wfes are rator and rand for the two components of a combination (i.e., its operator and its operand) and bv and body for the two components of a λ-expression (i.e., its

bound variable and its body). The terms are due to Landin [13]. As part of the tree presentation of a wfe, we may label the branches with these names



A path is simply a sequence of branch names. For example, rator•bv and body•body•rand are paths.

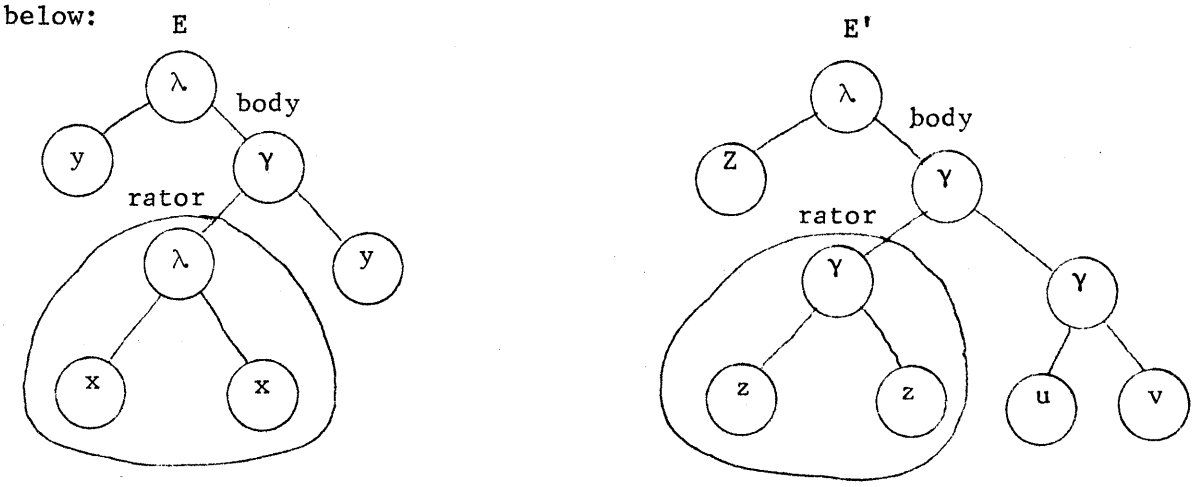
If b and c are paths, we say that b is a stem of c ($b \leq c$) if b is an initial segment of c. For example, rator $\leq$ rator•bv, and $\Lambda \leq$ b for any path b.

3. Subexpressions. In terms of the tree representation, a subexpression is simply a sub-tree,i.e., a certain node together with the branches and nodes depending from it. More precisely, a subexpression is a pair consisting of a path and an expression $< p , e >$ such that tracing the path from the top-most node of the whole expression leads to e. Two subexpressions $< p_1, e_1 >$ and $< p_2, e_2>$ of a given expression are disjoint if neither $p_1 \leq p_2$ nor $p_2 \leq p_1$. If $p_1 \leq p_2$, we say that $< p_2, e_2>$ is a part of $< p_1, e_1>$. For example, the subexpressions of

$(\lambda x.xx)$

are $<\Lambda , \lambda x.xx >$, $< bv, x >$, $< body, (xx) >$, $< body•rator, x>$ and $< body•rand, x >$. $< bv,x >$ and $< body•rator,x >$ are disjoint; and $< body•rator,x>$ is part of $< body, (xx)>$.

If S and S' are subexpressions of E and E', respectively, we say that S is homologous to S' if S occupies the same position in E that S' occupies in E'. Considering an expression as a tree, the position of S in E is simply the path in the tree from the topmost node of E to the topmost node in S. Consider the two wfes depicted below:



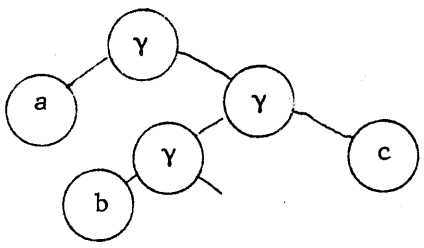
The two circled subexpressions are homologous since they have the same positions in E and E', namely, body•rator.

An occurrence of e in E is a subexpression < p, e > of E.

A context E[ ] is an expression with a "hole" in it. For example,

a(b [ ] c)

and



are presentations of the same context.

If E[ ] is a context, E[A] is derived by replacing the hole by A, where A is a wfe or another context. In the former case E[A] is a wfe; in the latter it is a context as it still contains a hole.

B. Conversion Rules

Having established the syntax of wfes, we now introduce a number of rules for transforming them.

Definition: If E and F are two wfes, E ≡ F means they are identical, the same wfe.

The free variables of a wfe are defined inductively as follows:

1. If E ≡ x, a variable, then x is a free variable of E.
2. If E is a combination (R D), then x is free in E iff x is free in R or x is free in D.
3. If E is a λ-expression (λy.M) then x is free in E iff x is free in M and x ≠ y.

A variable is bound in E if it occurs in E and is not free in E.

There are three basic conversion rules for wfes.

1. α-conversion. $E[\lambda x.M]$ may be converted to $E[\lambda y.M']$ if y is not free in M and M' results from M by replacing every free occurrence of x by y. We write $E[\lambda x.M] =_\alpha E[\lambda y.M']$. For example,

$$x(\lambda a.xa(\lambda a.a)) =_\alpha x(\lambda b.xb\ (\lambda a.a))$$

2. β-reduction. $E[(\lambda x.M)N]$ is β-reducible if no variable which occurs free in N is bound in M. (This proviso prevents the "capturing" of free variables.) Specifically, $E[(\lambda x.M)N]$ is reducible to $E[M']$ where M' results from the replacement of all free occurrences of x in M by N. We write $E[(\lambda x.M)N] >_\beta E[M']$. For example,

$$(\lambda x.(\lambda y.xy)x) >_\beta (\lambda x.xx)$$

It should be clear that any wfe of the form $(\lambda x.M)N$ can be made β-reducible by a series of α-conversions which change the bound variables of M so as to meet the requirement stated above. In fact, we can define a function on expressions, subst, such that

$$(\lambda x.M)N > \text{subst}[N,x,M]$$

where we take > to mean convertible by a combination of α-conversions and β-reductions.

subst[N,x,M] ≡

(i) if M ≡ x then N

(ii) if M is a variable y ≠ x then y

(iii) if M is a constant, then M.

(iv) if M is a combination (RD), then

(subst[N,x,R] subst[N,x,D])

(v) If M is a λ-expression $(\lambda y.\ K)$

(a) if y ≡ x then M

(b) if y ≠ x and y is <u>not</u> free in N, then

$(\lambda y.\ \text{subst}[N,x,K])$

(c) if $y \neq x$ and y is free in N then

$(\lambda z.\ \text{subst}[N,x,\ \text{subst}[z,y,K]])$ where

z is not free in K and $z \neq x$.

This definition is from Curry [11].

3. η-reduction. A wfe $E[(\lambda x.Mx)]$ where x is not free in M is reducible ($>_\eta$) to E[M]. We include this kind of reduction primarily for completeness; it does not figure prominently in the sequel.

4. δ-reduction. In addition to the foregoing conversion rules, we allow for an indefinite class of rules called δ-rules. They shall always be associated with the constants that appear in a particular system. Each δ-rule has the form

"E may be replaced by E' in any context."

There are several restrictions on E and E'.

(1) E has the form $((cA_1)\ldots A_N)$ where c is a constant.

(2) E contains no free variables.

(3) No proper subexpression of E is reducible by rules β, or any other δ-rule.

(4) E' contains no free variables.

5. Terminology. We now introduce several terms related to conversion and reduction.

(1) A redex is any expression which is reducible by rules β, η, or δ (possibly after some α-conversions)

(2) Abstraction is the inverse operation from reduction. If $A > B$, we can say that A comes from B via an abstraction. We may also write $B < A$.

(3) We write $A \geq B$ if A is reducible or α-convertible to B.

(4) We define $A = B$ inductively as follows:

$A = B$ iff (1) $A \geq B$ or $B \geq A$

or (2) there exists a C such that $A=C$ and $C=B$

For emphasis, we sometimes say A and B are inter-convertible when $A=B$.

(5) A wfe is in normal form if it contains no redex as a subexpression.

Any of these terms can be specialized to a particular kind of conversion rule. For instance, a wfe is in β-normal form if it contains no β-redex.

Example 1. Not every wfe is reducible to a normal form

$$(\lambda x.xx)(\lambda y.yy) > (\lambda y.yy)(\lambda y.yy)$$
$$> (\lambda y.yy)(\lambda y.yy) > \ldots.$$

Example 2. A specific set of constants and δ-rules may be employed to construct a λ-calculus applicable to a particular domain of discourse. For example, a primitive system dealing with the natural numbers might be the following:

(a) constants: 0, suc, pred, zero

(b) δ-rules: an expression is a numeral if it has the form $(\text{suc}^n 0)$; e.g., 0, suc 0, suc (suc 0) are numerals. Then the (infinite) set of rules are the following:

if x is a numeral

(1) (pred 0) may be replaced by 0

(2) (pred (suc x)) may be replaced by x

(3) (zero 0) may be replaced by true ≡(λa.λb.a)

(4) (zero (suc x)) may be replaced by false≡(λa.λb.b)

Example 3. An interesting wfe, which we shall deal with extensively later, is

$$Y \equiv \lambda f.(\lambda h.f(hh))(\lambda h.f(hh))$$

Y represents a solution to the following general problem: Given any wfe, F, find another wfe, A, such that

$$A = (FA)$$

(where = means convertible to)

Interpreting F as a function, we call A a fixed-point of F because F maps A into A.

It happens that every wfe F has at least one fixed point and applying Y to F produces it! That is, for any F

$$(YF) = (F(YF))$$

For, $YF \equiv (\lambda f.(\lambda h.f(hh))(\lambda h.f(hh)))F$

$> (\lambda h.F(hh))(\lambda h.F(hh))$

$> F((\lambda h.F(hh))(\lambda h.F(hh)))$

$< F(YF)$

Y is called the paradoxical combinator by Curry who employs it in an explication of Russel's paradox [11]. Y is also reminiscent of the way Gödel's substitution function is employed in the proof of his incompleteness theorem although Theorem 3 of this chapter

appears more anaolgous to Gödel's argument. See Rogers [12 , p. 202] for further discussion on this point.

Our main interest in Y is its use in constructing wfes defined by recursive equations. For example, using the constants of Example 2 we seek a wfe sum such that

$$\text{sum} = \lambda x.\lambda y.\ \text{zero}\ x\ y\ (\text{suc}(\text{sum}(\text{pred}\ x)y))$$

$$< (\lambda f.\lambda x.\lambda y.\ \text{zero}\ x\ y\ (\text{suc}\ (f(\text{pred}\ x)y)))\ \text{sum}$$

Thus, a solution is sum $\equiv$ YF where $F \equiv \lambda f.\lambda x.\lambda y.\text{zero}\ x\ y(\text{suc}(f(\text{pred}\ x)y))$ Notice that, if we define $\bar{n} \equiv \text{suc}^n 0$,

$$\text{sum}\ \bar{n}\ \bar{m} > \overline{n + m}$$

For example,

$$\begin{aligned}
\text{sum}\ \bar{1}\ \bar{2} &> \text{zero}\ \bar{1}\ \bar{2}\ (\text{suc}(\text{sum}(\text{pred}\ \bar{1})\bar{2})) \\
&> (\lambda a.\lambda b.b)\bar{2}(\text{suc}\ \ldots) \\
&> (\text{suc}(\text{sum}\ \bar{0}\ \bar{2})) \\
&> (\text{suc}\ (\text{zero}\ \bar{0}\ \bar{2}(\ \ldots\ ))) \\
&> (\text{suc}((\lambda a.\lambda b.a)\bar{2}(\ldots))) \\
&> \text{suc}\ \bar{2} \\
&\equiv \bar{3}
\end{aligned}$$

Notice how $(\lambda a.\lambda b.a)$ is used to "throw away" part of an expression. This is the technique used to achieve "conditional branching."

C. Basic Theorems

We now state, without proof, several basic results about conversion. Before doing so, we must introduce certain supporting concepts associated with the conversion rules.

1. The Father Function

If E' is the result of a transformation of E, we shall define a function, father, from subexpressions of E' to subexpressions of E. The relation, son, is the inverse of father, and the relations descendant and ancestor are their natural extensions by transitivity. Roughly speaking, each son is a copy of its father. We shall define father for each kind of expression conversion.

Let S' be a subexpression of E'.

(a) If E' is the result of an α-conversion of E, then the father of S' is the homologous subexpression in E.

(b) If E' arises from a β-reduction on some subexpression $(\lambda x.M)N$ of E, then we consider three cases:

(1) If S' is not part of the contractum (i.e., subst[N,x,M])of $(\lambda x.M)N$, then its father is the homologous subexpression in E.

(2) If S' is part of an occurrence of N which was substituted for x in M, then its father is the homologous part of the occurrence of N in $(\lambda x.M)N$.

(3) If S' is any other part of subst[N,x,M] then its father is the homologous subexpression in the occurrence of M in $(\lambda x.M)N$.

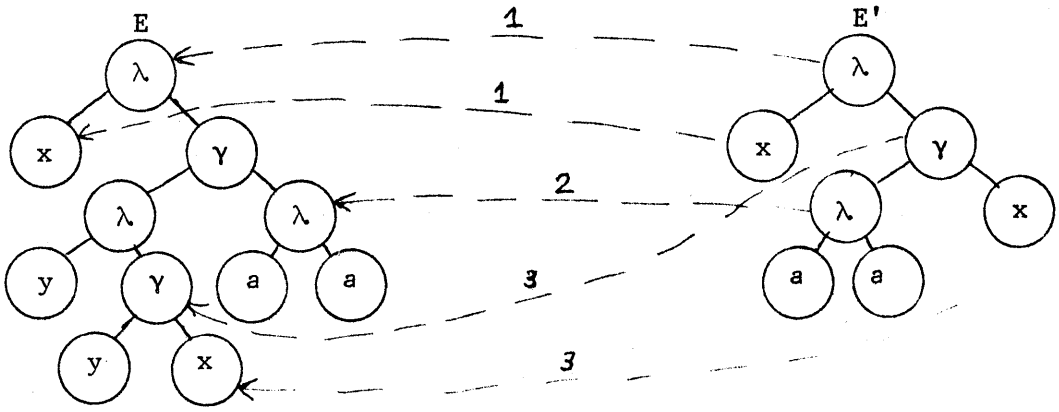
Notice that neither the redex $(\lambda x.M)N$ nor any of the free occurrences of x in M have any sons in E'. Also, if there are k free occurrences of x in M, every subexpression of N will have k sons in E.

Example 4. Consider the expressions:

$$E \equiv \lambda x.(\lambda y.yx)(\lambda a.a)$$

$$E' \equiv \lambda x.(\lambda a.a)x$$

Some son-to-father relations are shown in the figure below.



Each dotted son-to-father arrow has been labeled with the number of the clause above which defines it.

(c) If E' arises from an η -reduction of a subexpression $(\lambda x.Mx)$ in E, we consider two cases:

(1) If S' is part of the contractum, M, then its father is the homologous subexpression in M

(2) If S' is not part of the contractum, its father is the homologous subexpression in E.

Notice that neither $(\lambda x.Mx)$ nor Mx has any sons in E'.

(d) If E' arises from E by a δ-reduction of $(c\ A_1 \ldots A_N)$ and S' is not a part of the contractum, then its father is the homologous subexpression in E. A subpart of the contractum has no father in E unless otherwise specified for the particular rule.

The notion of a descendant is a minor generalization of a residual as defined by Church [1] and Curry [11]. Specifically, the descendant of a redex may be called a residual.

2. Restricted Reductions

Let R be a set of redexes in a wfe E and consider a series of reductions on E in which every redex contracted is a descendant of a member of R. We call such a reduction sequence a reduction relative to R. A complete reduction relative to R is one in which the final wfe contains no descendants of R. For example, consider

$$A \equiv \underset{1}{(\lambda x.(\lambda a.ax)\ (\lambda c.x))}\underset{2}{(\lambda y.\ y\ b)}$$

which contains two underlined β-redexes, 1 and 2. We have two reductions relative to {1, 2}

$$A > \underset{1}{(\lambda x.(\lambda c.\ x)x)\ (\lambda y.yb)}$$

$$> (\lambda c.(\lambda y.yb))(\lambda y.yb) \equiv B$$

$$\text{and } A > \underset{2}{(\lambda a.a(\lambda y.yb))(\lambda c.(\lambda y.yb))}$$

$$> B$$

Notice that the redex in B is a descendant of (ax) in A and cannot be contracted in a reduction relative to {1, 2}.

The following Lemma is fundamental and is sometimes called the lemma of parallel moves.

Lemma 1. (Curry)

Let R be a set of β and δ redexes in B. Then

(a) Any reduction of B relative to R can be extended to a complete reduction of B relative to R.

(b) Any two complete reductions of B relative to R end on the same wfe, C, and

(c) Any part of C has the same ancestor in B (if any) regardless of which reduction sequence is used.

This Lemma roughly corresponds to Curry's Property (E) and its proof is easily derived from his proof in [11, Chapter 4].

The Lemma is not true if -redexes are considered as shown by Curry's example

$$\underline{\underline{(\lambda x.\ (\lambda y.z)x)N}}_1$$
$$2$$

Contraction of the β-redex, 1, yields

$$(\lambda y.z)N.$$

which contains no descendents of {1, 2}, while contraction of the -redex, 2, yields

$$\underline{(\lambda y.z)N}_1$$

which contains a son of 1. Thus a complete reduction relative to {1, 2} yields (λy.z)N if 1 is contracted first but yields z if 2 is contracted first.

Notice that if η-reductions are not permitted, the descendant of a redex is always a redex.

Lemma 1 is basic to the proof of the Church-Rosser Theorem as proved in Curry [11].

Theorem 1. (Church-Rosser)

If X = Y then there exists a

Z such that

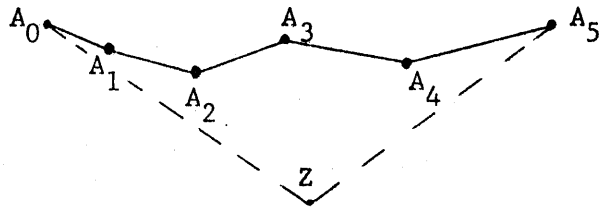
$X \geq Z$ and $Y \geq Z$

The proof of this theorem constitutes Chapter 4 of [11].

A natural way to understand the meaning of the theorem is to visualize the reduction process as a line proceeding from left to right in which downward sloping lines represent reductions and upward lines represent abstractions. For example, if we have

$$A_0 \geq A_1 \geq A_2 \leq A_3 \geq A_4 \leq A_5$$

The passage from A_0 to A_5 can be depicted as shown below.



Theorem 1 states that the transformation from A_0 to A_5 can be made so that there is just one downward sloping line followed by one upward, as shown by the dotted lines above.

Corollary 1. If x=y and y is in normal form, then $x \geq y$. If x is also in normal form, then $x =_{\alpha} y$.

Thus, if we are interested in reducing wfes to normal form, it does not matter precisely how a given wfe is brought to normal form; the final expression will be the same regardless of how it was derived. This is not to say that the choice of reductions is completely unimportant. There are wfes which can be reduced to normal form but which also can be reduced in such a way as to prolong the reduction indefinitely. For example,

$$A \equiv (\lambda x.a)((\lambda x.xx)(\lambda x.xx))$$

$$> A \quad > A$$

But $A >$ a also,

While trying all possible reduction sequences will reduce a wfe to normal form whenever one exists, there are other algorithms which are more direct. If S and S' are two subexpressions of a wfe, E, we say that S is to the left of S' if its beginning is to the left of the beginning of S' when E is written linearly. If A and B are β and δ-redexes, notice that either A is to the left of B or B is to the left of A or they are the same redex.

We define a standard order reduction.

$$A_0 > A_1 > A_2 > \ldots > A_N$$

as one in which the passage from A_i to A_{i+1} is effected by contraction of the left-most β or δ redex in A_i.

Theorem 2. (Curry)

If a wfe, A, has a β-δnormal form, then a standard order reduction beginning with A yields that normal form.

This result is easily derivable from the proof of Curry's standardization theorem.

D. λ-Expressions Gödelized

In order to demonstrate that certain problems related to the λ-calculus are formally unsolvable, we shall develop a formalization of the λ-δ-calculus within itself. This technique was presented by Scott [14].

1. Natural Numbers. As a preliminary we choose a representation of the natural numbers as λ-expressions. There are many possibilities; we choose one similar to Church's in [1]. The representation, n*, of the number n is

$$n^* \equiv \lambda a.\lambda b.a^n b$$

Thus,

$$0^* \equiv \lambda a.\lambda b.b$$

$$1^* \equiv \lambda a.\lambda b.ab$$

$$2^* \equiv \lambda a.\lambda b.a(ab)$$

$$3^* \equiv \lambda a.\lambda b.a(a(a\ b))$$

etc.

The functions, suc, pred, and eq, can be defined as λ-expressions so that for any n = 0, 1, ...

$$\text{suc } n^* = (n+1)^*$$

$$\text{pred } 0^* = 0^*$$

$$\text{pred } (n+1)^* = n^*$$

$$\text{eq } m^*\ n^* = \text{true if } m=n, \text{ false otherwise}$$

where true $\equiv \lambda x.\lambda y.x$, false $\equiv \lambda x.\lambda y.y$

The definitions of these functions can be found in Böhm [15].

2. Quotations. Suppose that the variables and constants of a λ-δ-calculus are chosen from the infinite lists

$$x_0, x_1, \ldots, x_i, \ldots$$

and $$c_0, c_1, \ldots, c_i, \ldots$$

respectively. Clearly this constitutes no formal restriction on the system.

We define the quotation, E*, of any wfe, E, inductively as follows:

$$x_i^* \equiv \lambda x_0.\lambda x_1.\lambda x_2.\lambda x_3.x_0 i^*$$

$$c_i^* \equiv \lambda x_0.\lambda x_1.\lambda x_2.\lambda x_3.x_1 i^*$$

$$(R\ D)^* \equiv \lambda x_0.\lambda x_1.\lambda x_2.\lambda x_3.x_2\ R^*D^*$$

$$(\lambda V.B)^* \equiv \lambda x_0.\lambda x_1.\lambda x_2.\lambda x_3.x_3\ V^*B^*$$

Thus, quotation is a one-to-one mapping of the set of all wfes to a subset of them, call it Q. Note that every wfe in Q is in normal form, and that $x^* = y^*$ implies $x \equiv y$. Notice, also, that no member of Q contains a constant.

The question naturally arises whether there is some wfe, quote, such that quote x = x* for any wfe x. The answer is no, by the following argument:

Suppose $x = y$ but $x \not\equiv y$; i.e., x and y are interconvertible but not identical. Then (quote x) = (quote y), which implies $x^* = y^*$. But then $x \equiv y$, contradicting the supposition.

Now we make a claim, vital to the validity of any undecidability results.

Any effective process or decision procedure that a person (or machine) can carry out on wfes can be described by a wfe which operates on quotations.

For example, if there were a way of deciding whether any wfe can be reduced to normal form, we claim that there would be a particular wfe, normal such that

$$(\text{normal } x^*) = \begin{cases} \text{true if x has a normal form} \\ \text{false otherwise} \end{cases}$$

To support this claim, we shall write a simple λ-calculus function to determine whether a quotation represents a wfe which is in β-normal form. Specifically,

$$\text{norm } x^* = \begin{cases} \text{true if x is in } \beta\text{-normal form} \\ \text{false otherwise} \end{cases}$$

We define norm recursively as follows:

$$\begin{aligned} \text{norm} = \lambda x.x(&\lambda a.\ \text{true}) \\ &(\lambda a.\ \text{true}) \\ &(\lambda r.\lambda d.r(\lambda a.\ \text{norm d}) \\ &\qquad (\lambda a.\ \text{norm d}) \\ &\qquad (\lambda r'.\lambda d'.\ \text{norm r (norm d) false}) \\ &\qquad (\lambda v.\lambda b.\ \text{false})) \\ &(\lambda v.\lambda b.\ \text{norm b}) \end{aligned}$$

Since norm is expected to work only on quotations, we know that x will be a wfe of the form $\lambda x_0.\lambda x_1.\lambda x_2.\lambda x_3.x_i\ldots$, for i=0,1,2, or 3.

Thus, x will select one of the four expressions arrayed vertically and apply it to whatever follows x_i. The algorithm can be described as follows: To check x for normal form, see if x is a constant or variable, in which case the answer is true. If x is a λ-expression, check its body for normal form. If x is a combination, consider its rator. If the rator is a λ-expression, the answer is false. If the rator is a constant or variable, check the rand. Otherwise the rator is a combination; check the rator and the rand - both must be in normal form.

Other functions to deal with quotations are straightforward but tedious to write. We shall list some useful functions without supplying the details of their construction.

aconv x* y* = true if x and y are α-convertible;
false otherwise.

reduce x* = y* where y results from x by a single contraction of the left-most β- or δ-redex of x. We assume that the rules associated with constants can be formalized.

eval x* = y* where y results from a standard reduction of x, if x has a β-δ normal form. Otherwise, eval x* itself has no normal form.

quote x* = (x*)*

For example,

$$(x_0{*})* = (\lambda x_0.\lambda x_1.\lambda x_2.\lambda x_3.x_0 0*)*$$
$$= \lambda x_0.\lambda x_1.\lambda x_2.\lambda x_3.x_3 \quad (x_0)* \ (\lambda x_1.\lambda x_2.\lambda x_3.x_0 0*)*$$

etc.

Thus, quotations play a role analogous to Gödel numbers in other formal theories.

Theorem 3. (Scott)

For any wfe, F, there exists a wfe, A, such that

$$A = FA^*$$

Proof: Let $H \equiv \lambda x.F(\text{Makec } x(\text{quote } x))$

when Makec $\equiv \lambda a.\lambda b.\lambda x_0.\lambda x_1.\lambda x_2.\lambda x_3.x_2\ a\ b$

(i.e., Makec constructs a quoted combination from two already quoted expressions)

Then, let $A \equiv H\ H^*$

$A > F\ (\text{Makec } H^*(\text{quote } H^*))$

$> F(\text{Makec } H^*(H^*)^*)$

$> F((H\ H^*)^*)$

$\equiv F\ A^*$

QED

This theorem is an analogue of Kleene's second recursion theorem. Its use in proving undecidability results is brought out by the following:

Theorem 4. (Scott)

Let B be any non-empty, proper subset of all the wfes. Then if $x=y$ implies $x \in B \Leftrightarrow y \in B$, then it is undecidable whether a wfe is contained in B.

Proof:

Suppose we have a predicate, P, such that

$Px^* =$ true if $x \in B$, false otherwise.

Then choose some $b \in B$ and $a \notin B$ and define

$F \equiv \lambda x.Pxab$

By Theorem 1 there is a z such that

$z = Fz^*$

$= Pz^*ab$

$=$ a if $z \in B$, otherwise b

But $z \in B \wedge z=a$ implies $a \in B$

and $z \notin B \wedge z=b$ implies $b \notin B$

It follows from the contradiction that P cannot exist. Therefore, by the claim, we cannot decide membership in B.

QED

Corollary 4 It is undecidable whether a wfe has a normal form.

Proof:

The set, N, of wfes having normal form, is such that

$x=y$ implies $x \in N \Leftrightarrow y \in N$

QED

E. λ-calculus as a Model of a Programming Language

The λ-calculus provides a simple model of the variable and subroutine facilities found in languages of the ALGOL variety. Landin's work in "A Correspondence between ALGOL-60 and Church's λ-calculus" [4] illustrates this point in detail.

In this section we endeavor to construe the λ-calculus as a programming language in its own right. Obviously, it lacks many of the important features found in real programming languages (most notably, destructive assignment and jumping). Nevertheless, it has enough in common with real languages to give relevance to many of its interesting properties.

We now identify several informal concepts common to programming languages and note how they correspond to features of the λ-calculus.

1. The Computer. The computer for λ-expressions is simply an expression reducer or evaluator. As input this computer accepts a wfe and attempts to reduce it to a normal form using the rules of α, β, η conversion and (possibly) some rules of δ-conversion. If it ever succeeds, it prints the resulting normal form wfe as output. If the wfe cannot be reduced to normal form, the computer never halts.

The specific way in which the computer operates need not be specified further since, by Corollary 1, the final normal form does not depend upon which way reductions are carried out. We may assume, if we wish, that the computer performs reduction in standard order; but any

technique which assures getting to a normal form when one exists will do as well.

2. Domain of Discourse. At first glance it appears that the λ-calculus is a language with no data objects, such as numbers or strings, to talk about. However, these are easily introduced by constants and δ-rules. For example, the constants and δ-rules discussed in section B provide a language having natural numbers as its domain of discourse. It is also possible to simulate familiar data objects with wfes themselves as in Section C, but we shall not pursue that possibility here.

3. Program vs. Data. The distinction between programs and data is thoroughly muddled by this system, but we can interpret a combination (P D) submitted to the computer as asking it to apply program P to the datum D. In a similar way, we might interpret $((\lambda f_1.\lambda f_2.P)F_1F_2\ D)$ as a request that the computer apply the program P, which uses F_1 and F_2 as subroutines, to the datum D. Officially, though, the computer is just an expression reducer and such interpretations are strictly in the mind of the beholder.

4. Conditional Branches. The ALGOL-60 statement if P then A else B can be simulated by the combination (P A B) where we expect that the expression P will always reduce to true ≡ $(\lambda x.\lambda y.x)$ or false ≡ $(\lambda x.\lambda y.y)$.

5. Loops. Repetitive operations can be achieved by the use of recursion as discussed in section B.

6. Assignment. The assignment of a value to a variable for the first time is simply accomplished. To simulate the ALGOL-60 sequence

x:= 1;

S

where S does not re-assign a new value to x, we write

$(\lambda x.S)1$

This is the closest thing to assignment available.

7. Functions.

To define the function f(x) = x+1, and use it to compute f(3)* f(4) we write

$(\lambda f.f(3)*f(4))(\lambda x.x+1)$

CHAPTER III

Recursion

The notion of recursion is useful in computer programming. There are many well-known examples of recursive algorithms. Recursion often allows a succinctness not easily achieved by other means. In this chapter we shall explore the mechanism of recursion by presenting an analogue of Kleene's first recursion theorem for the λ-calculus. While Kleene has emphasized the role of this theorem as evidence for Church's thesis, our main interest in it is due to the fact that it brings out a relationship between purely formal computation and the more informal notion of solving an equation.

A. Two Views of Recursion

Consider the statement

$$f(m,n) = \underline{\text{if}}\ m=n\ \underline{\text{then}}\ n+1\ \underline{\text{else}}\ f(m,f(m-1,n+1)) \qquad (1)$$

How can we interpret it as a definition of the function f? Ostensibly, it appears to be a statement about f and some numbers, m and n, hardly a definition.

1. Formal Computation

Equation (1) can be used to deduce the value of f(m,n) for various integers m and n. Let us introduce some formal deduction

rules to illustrate how this might be done. Each rule tells how to deduce a new equation from some existing ones.

I. If x and y are numerals, x+y may be replaced by the numeral denoting the sum of x and y, and x-y may be replaced by the numeral denoting the excess of x over y (possibly negative).

II. If x and y are the same numeral, "if x=y then E_1 else E_2" may be replaced by E_1. If they are distinct numerals, it may be replaced by E_2. E_1 and E_2 may be any expressions.

III. From a given equation containing a variable, V, we may deduce a new equation by substituting a numeral for V throughout the equation.

IV. Given (1) an equation of the form $f(a_1,\ldots, a_N)=b$ where $a_1,\ldots, a_N$ and b are numerals, and (2) an equation $E_1=E_2$ where neither E_1 nor E_2 contain any variables and E_2 contains an occurrence of $f(a_1,\ldots, a_N)$: we may derive a new equation $E_1=E_2'$ where E_2' arises from E_2 by replacement of $f(a_1,\ldots, a_N)$ by b.

Rules III and IV are Kleene's rules R1 and R2 for formal computation with equation schemata. In Kleene's system [16], Rule I is simplified and Rule II is not used, as his system does not use if expressions. Nevertheless, the system presented here is roughly equivalent to his.

Now let us compute f(2,0) by attempting to deduce f(2,0)=k for some k, using Rules I through IV.

1.	f(m,n) = <u>if</u> m=n <u>then</u> m+1 <u>else</u> f(m,f(m-1,n+1))	(Given)
2.	f(1,1) = <u>if</u> 1=1 <u>then</u> 1+1 <u>else</u>....	(III applied to 1)
3.	f(1,1) = 2	(I, II on 2)
4.	f(2,0) = <u>if</u> 2=0 <u>then</u> 2+1 <u>else</u> f(2,f(2-1,0+1))	(III on 1)
5.	f(2,0) = f(2,f(1,1))	(I, II on 4)
6.	f(2,0) = f(2,2)	(IV on 3 & 5)
7.	f(2,2) = <u>if</u> 2=2 <u>then</u> 2+1 <u>else</u>....	(III on 1)
8.	f(2,2) = 3	(I, II on 7)
9.	f(2,0) = 3	(IV on 6 & 8)

Notice that we have a certain degree of latitude about the order in which things are proved. It might be possible that certain sets of equations would allow us to deduce different values for a function depending upon which order of deduction we used. In such a case we say that the equations are <u>inconsistent</u>. For example,

$$f(x)=x+1$$

$$f(x)=x$$

would be such a set. We claim the set consisting of just equation (1) is not inconsistent.

Now we decree that the partial function defined by (1) is the set of all pairs $< <m,n >, k >$ such that f(m,n) = k is deducible from equation (1) using the deduction rules. In general, a consistent set of equations, E, defines a partial function f of n arguments as the set of all pairs $< < x_1,\ldots,x_n>, y>$ such that $f(x_1,\ldots,x_n) = y$ is deducible from E using the rules of deduction.

Thus, we have given one interpretation of (1) as a definition.

2. The Minimal Solution

Another possibility is to view (1) as a statement which may be true or false for a particular partial function, f. In other words, we assume that f is a single-valued[+] subset of (N×N)×N and interpret (1) as requiring:

(a) for all $m \in N$, $\langle\langle m,n\rangle, m+1\rangle \in f$

(b) for all $m, n \in N$, such that $m \neq n$, $\langle\langle m,n\rangle, k\rangle \in f$ if and only if there exists a k' such that $\langle\langle m-1,n+1\rangle, k'\rangle \in f$ and $\langle\langle m,k'\rangle, k\rangle \in f$. That is: either $f(m,n)$ and $f(m,f(m-1,n+1))$ are both undefined or they are both defined and equal.

There are many such f's that satisfy (a) and (b). For example,

$f_1 = \{ \langle\langle m,n\rangle, m+1\rangle \mid m \in N \wedge n \in N \}$

$f_2 = \{ \langle\langle m,n\rangle, m+1\rangle \mid m \geq n \} \cup \{ \langle\langle m,n\rangle, n-1\rangle \mid m < n \}$

$f_3 = f_1 \cap f_2 = \{ \langle\langle m,n\rangle, m+1\rangle \mid m \geq n \}$

$f_4 = \{ \langle\langle m,n\rangle, m+1\rangle \mid m \geq n \text{ and } (m-n) \text{ is even} \}$

$f_5 = \{ \langle\langle m,n\rangle, n-1\rangle \mid m < n \text{ and } H(m,n) \} \cup$
$\{ \langle\langle m,n\rangle, m+1\rangle \mid m \geq n \text{ or not } H(m,n) \}$

where $H(m,n)$ is true iff Turing machine $|m|$ with input $|n|$ halts. Notice that f_5 is not a computable function; if it were, we could use it to solve the halting problem.

As an example, consider f_4. Clause (a) is obviously satisfied. To show clause (b) holds in one direction, we argue as follows:

Suppose for some $m \neq n$, $\langle\langle m,n\rangle, k\rangle \in f_4$. Then $k=m+1, m > n$, and $m-n$ is even. Hence $m-1 \geq n+1$ and $(m-1)-(n+1)$ is even, so $\langle\langle m-1,n+1\rangle, m\rangle \in f_4$.

[+] A set of pairs, B, is single-valued if $\langle x,y\rangle \in B$ and $\langle x,y'\rangle \in B$ implies $y=y'$.

Since $\langle\langle m,m\rangle,m+1\rangle \in f_4$ the "only if" part of clause (b) is true with $k=m+1$ and $k'=m$.

The argument for the "if" part of clause (b) is analogous.

Obviously, equation (1) interpreted in this way does not constitute a definition of any single function. However, the first Kleene recursion theorem provides a way of choosing a single function from all the possibilities that satisfy (1). For our purposes the theorem can be stated as follows:

Theorem 1 (Kleene). Any equation of the form

$$f(x_1,\ldots,x_n)=E[f,x_1,\ldots,x_n]$$

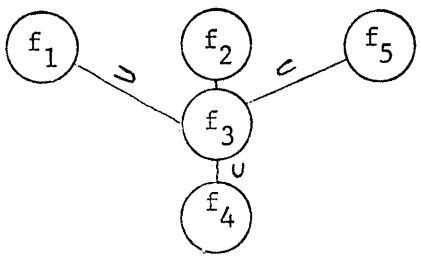
where $E[f,x_1,\ldots,x_n]$ is an expression built up from $f,x_1,\ldots,x_n$ using functional application and *if* clauses, has a solution for f, call it g, such that

(1) g is the function defined by the formal computation rules of section 1 and therefore is partial recursive (i.e., computable)

and (2) if g' is another solution, $g \subset g'$

i.e., g' is an extension of g.

In light of Theorem 1, we may decree that *the* function defined by (1) is the minimal solution to (1); i.e., the one which is contained in all other solutions. If we consider the inclusion relations which obtain among $f_1,\ldots,f_5$, we arrive at the relations depicted below:



It happens that there are no solutions properly contained in f_4, so it is the partial function defined by (1).

3. An Alternative Deduction Procedure

As a prelude to considering a recursion theorem for the λ-calculus, we shall consider a deduction system derived from the system of section 1 by a slight change.

Instead of Rule IV we use the following, more permissive rule:

IV'. Given an equation of the form $f(x_1,\ldots,x_N)=E$ where $x_1,\ldots,x_N$ are variables and E is any expression; and given an equation $r=s$ where s contains an occurrence of $f(A_1,\ldots,A_N)$ where $A_1,\ldots,A_N$ are any expressions; we may derive a new equation $r=s'$ where s' is constructed in two steps.

(1) for $i=1,\ldots,N$ replace all occurrences of x_i in E by A_i, deriving E'

(2) replace the occurrence of $f(A_1,\ldots,A_N)$ in s by E' deriving s'.

This rule roughly corresponds to β-conversion or ALGOL-60 call-by-name.

Does the function defined by a set of equations differ under this new set of deduction rules? Apparently, it does in the cases

where constant functions are involved. Consider the following equation:

$$f(m,n) = \underline{\text{if}}\ m=0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ 1+f(m-1,f(n-2,m)) \qquad (2)$$

If the new rule, IV', is to be used, we shall be able to deduce $f(x,y)=x$ for any numerals $x \geq 0$ and y. For example, to show $f(1,5)=1$, we proceed as follows:

1. $f(1,5) = \underline{\text{if}}\ 1=0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ 1+f(1-1,f(5-2,1))$ (by III)
2. $f(1,5) = 1+f(0,f(3,1))$ (by I, II)
3. $f(1,5) = 1+[\underline{\text{if}}\ 0=0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ 1+f(0-1,f(f(3,1)-2,0))]$ (by IV')
4. $f(1,5) = 1+0$ (by II)
5. $f(1,5) = 1$ (by I)

On the other hand, using the old rule, IV, we find that f is the partial function

$$f'(m,n)=\underline{\text{if}}\ m=0\ \underline{\text{then}}\ 0\ \underline{\text{else}}$$
$$\underline{\text{if}}\ m=1 \wedge n=2\ \underline{\text{then}}\ 1\ \underline{\text{else}}\ \underline{\text{undefined}}$$

It is easy to see that $f(0,n)=0$ (for any n) and $f(1,2)=1$ can be deduced by Rules I-IV. To show that no other values can be deduced, we need only show that

$$f' = \{ \langle\langle 0,n \rangle, 0 \rangle \mid n \in N \} \cup \{ \langle\langle 1,2\rangle, 1 \rangle \}$$

satisfies (2) in the sense of Theorem 1. That is:

(a) for all $n \in N$, $\langle\langle 0,n \rangle, 0 \rangle \in f'$

and (b) for all $m \neq 0, n \in N$: $\langle\langle m,n \rangle, k+1 \rangle \in f'$

if and only if there exists a k' such that

$\langle\langle n-2,m\rangle, k'\rangle \in f'$ and $\langle\langle m-1,k'\rangle,k\rangle \in f'$

To show (b) (only if)

$m \neq 0 \wedge \ll m,n \gg, k+1 > \epsilon f' \Rightarrow m=1 \wedge n=2 \wedge k=0$

$\Longrightarrow \ll n-2,m \gg, k' > = \ll 0,1 > , 0 > \epsilon f'$

and $\ll m-1,k' \gg,k > = \ll 0,0 > , 0 > \epsilon f'$

To show (b) (if)

$m \neq 0 \wedge \ll n-2,m > , k' > \epsilon f' \wedge \ll m-1,k' > , k > \epsilon f'$

$\Longrightarrow$ either $n-2=0 \wedge k'=0 \wedge m-1=0 \wedge k=0$

or $n-2=1 \wedge m=2 \wedge k'=1 \wedge m-1=0 \wedge k=0$

The second alternative is impossible, and the first implies

$$\ll m,n > , k+1 > = \ll 1,2 > , 1 > \epsilon f'$$

Therefore, f' is a solution to (2) and by Theorem 1, it must be the function defined by (2), given Rules I-IV.

Is there an analogue to Theorem 1 for the deduction Rules I-IV'? If we interpret functional equations in a slightly different way, we arrive at a possibility.

We may consider equation (2) to be a statement about an unknown <u>total</u> function from DXD to D where $D=N \cup \{\omega\}$ where ω is an object not in N. Intuitively, ω means undefined. Now, for such a function to satisfy (2), we must have the following:

(a) for all $n \epsilon D$, $f(0,n)=0$

(b) for all $m \neq 0$, $n \epsilon D$. $f(m,n)=1+f(m-1,f(n-2,m))$

Now the function

$f'(m,n)$ = <u>if</u> $m=0$ <u>then</u> 0 <u>else</u>

<u>if</u> $m=1 \wedge n=2$ <u>then</u> 1 <u>else</u> ω

does not satisfy (b). For example, $f'(1,0)=\omega$ while (b) requires that

$$f'(1,0)=1+f'(0,f'(-2,1))$$

Since $f'(-2,1) \epsilon D$, we have $f(1,0)=1+0=1$

We conjecture that the following analogue of Theorem 1 is true:

Theorem 2 (conjectured). If

$$f(x_1,\ldots,x_n)=E[f,x_1,\ldots,x_N]$$

is an equation as in Theorem 1, then if there is any solution for f, there is one, call it g, such that

(1) g is a total function on $D=N \cup \{\omega\}$

(2) g is computable by Rules I-IV'

(3) if h is any other total solution, then it is an extension of g in the sense that if $g(a_1,\ldots,a_n)=b\neq\omega$ for some $a_i \epsilon D$ then $h(a_1,\ldots,a_n)=b$

We feel that this theorem would follow from the minimal fixed-point theorem to be presented in a succeeding section. Specifically, if the equation schemata defining functions were translated into λ-δ-calculus formulae, the rule of β-conversion would serve as Rules III and IV', and the function denoted by a wfe would be the one singled out by Theorem 2.

As an example, let us translate equation (2) into a λ-δ-calculus equality. (We assume that the rules associated with if expressions and arithmetic expressions are given by δ-rules.)

$$f=Ff \qquad (3)$$

where $F\equiv \lambda g.\lambda m.\lambda n.$ if $m=0$ then 0 else $1+g(m-1)(g(n-2)m)$

Now if we interpret = to mean interconyertible, we know that YF satisfies (3). It turns out that YFxy reduces to x if x is a numeral ≥ 0, but has no normal form if $x<0$.

A simpler λ-calculus example is given by the following:

$$f=(\lambda x.xxx)f \qquad (4)$$

Three solutions for f are $Y(\lambda x.xxx)$, $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$, as may be easily verified. See Wozencraft [25] for further examples.

In Section C we shall show that the solution given by Y is the minimal one in the sense that it will give rise to a normal form in the fewest possible contexts. Before doing so, however, we develop the supporting concept of extensional equivalence.

B. Extensional Equivalence

There does not seem to be any natural, set-theoretic way to interpret a wfe as a partial function. To motivate our definition of extensional equivalence, we present a somewhat contrived interpretation.

Consider the machine, presented in Chapter II, Section E, which reduces wfes to normal form. We shall interpret a wfe as a partial function on wfes as follows: The function of n arguments ($n\geq 1$) denoted by the wfe F is defined as the set of all pairs $\langle\langle x_1,\ldots,x_n\rangle, y\rangle$ such that the machine, when given the wfe $(Fx_1x_2\ldots x_n)$ as input, halts and give the wfe y as output. Thus, if $\phi_n(E)$ is taken as the interpretation of E as a function of n arguments

$$\phi_n(F) = \{\langle\langle x_1,\ldots,x_n\rangle, y\rangle \mid (Fx_1\ldots x_n) \text{ has a normal form } y\}$$

It is clear that two wfes may denote the same function but still not be interconvertible. For example, $(\lambda x.xx)(\lambda x.xx)$ and $(\lambda x.xxx)(\lambda x.xx)$ are not interconvertible, but both denote the null function of n arguments, for any n.

We shall now develop a relation between wfes which leads to a definition of extensional equivalence.

Definition: A extends B, written $A \supset B$, if for all contexts E[ ]: if E[B] has a normal form then E[A]=E[B]; i.e., E[A] is convertible to E[B] and hence has the same normal form. (See Chapter II, Section A for the definition of a context.)

Intuitively, $A \supset B$ means that A is a "better" wfe than B, for we may replace B by A in any context and be assured of getting the same answer (i.e., normal form) if we were going to get an answer by using B. In addition, we may get an answer in certain contexts where using B would not provide an answer.

Theorem 3. If $A \supset B$ then, for any n, $\phi_n(A)$ is a functional extension of $\phi_n(B)$.

Proof: If we consider just these contexts, E[ ] where $E[F] = Fx_1\ldots x_n$

for some wfe's $x_1,\ldots,x_n$ we see that functional extension is simply a special case of wfe extension

QED

Henceforth, we shall consider only wfe extension as defined above.

Obviously, $\supset$ is reflexive and transitive. Furthermore, it is monotonic in the sense of the following:

Theorem 4. If $A \supset B$ then for any context $E[\ ]$, $E[A] \supset E[B]$

Proof: Assume $A \supset B$ and let $E_1[\ ]$ be any context. Let $E_2[\ ]$ be any context. Then $E_2[E_1[\ ]] \equiv E_3[\ ]$ is a context, and $E_3[B]$ having a normal form implies $E_3[A]=E_3[B]$. Thus, since $E_2[\ ]$ was chosen arbitrarily, we have $E_1[A] \supset E_1[B]$

QED

It follows from the Church-Rosser theorem that $A=B$ implies $A \supset B$ and $B \supset A$. Further, if B has a normal form, and $A \supset B$, then $A=B$. On the other hand, $A \supset B$ and $B \supset A$ does not imply $A=B$; we therefore make the following

Definition: $A \approx B$ iff $A \supset B$ and $B \supset A$.

Obviously, $\simeq$ is transitive, reflexive and symetric. Thus, it is an equivalence relation.

Intuitively, $A \simeq B$ means that it makes no difference whether A or B is used in any interpretation where we are ultimately interested only in normal forms. If either E[A] or E[B] has a normal form, then E[A]=E[B].

Let us briefly review the three kinds of equivalence possible between two wfes.

1. Identity. We say $A \equiv B$ if A and B denote identical wfes. For example:

$$x \equiv x$$

$$(\lambda y.a) \equiv (\lambda y.a)$$

$$(\lambda x.x) \not\equiv (\lambda y.y)$$

2. Convertibility. We say A=B if A can be converted into A' by a series of α, β and η-conversions, and $A' \equiv B$. Thus

$$(\lambda x.x) = (\lambda y.y) \qquad (\alpha\text{-conversion})$$

$$(\lambda a.a)b = b \qquad (\beta\text{-conversion})$$

$$(\lambda x.xx)(\lambda x.xx) \neq (\lambda x.xx)(\lambda x.xx)(\lambda x.xx)$$

3. Extensional Equivalence. We say $A \simeq B$ if the wfes, A and B, extend one another

Example: $(\lambda x.xx)(\lambda x.xx) \simeq (\lambda x.xxx)(\lambda x.xx)$

It is natural to ask whether there are larger, non-trivial equivalence relations than $\simeq$ which are monotonic in the sense of Theorem 4. The following theorem, due to Böhm [24] allows us to answer the question with a qualified negative for a pure λ-calculus without δ-rules.

Theorem 5. (Böhm) If A and B are in normal form, and $A \not= B$, then there exists a context E[ ] such that E[A]=C and E[B]=D for any arbitrary C and D. (= here means α, β, η -convertible)

Corollary 5. Suppose $\sim$ is an equivalence relation such that

(a) Non-trivial (i.e., there exist C, D such that $C \not\sim D$)

(b) An extension of = (i.e., A=B implies $A \sim B$)

(c) Monotonic (i.e., $A \sim B$ implies $E[A] \sim E[B]$)

(d) Normal (i.e., $A \sim B$ implies either both A and B have normal forms or both do not)

Then $A \sim B$ implies $A \simeq B$.

Proof: Suppose that there is an equivalence relation $\sim$ obeying (a) - (d). Suppose $F \not\sim G$. By the definition of $\simeq$, there exists an E[ ] such that either

(i) E[F] has normal form and E[G] does not

(ii) E[G] has normal form and E[F] does not

(iii) E[F] and E[G] have normal form and $E[F] \not\sim E[G]$

In case (i) and (ii) we have $E[F] \not\sim E[G]$ by property (d); hence $F \not\sim G$ by property (c).

In case (iii), choose any C, D, such that $C \not\sim D$ (by Property (a)). By Theorem 5 there exists an E' such that E'[E[F]] = C and E'[E[G]] = D. Therefore, by property (c), $F \not\sim G$.

Thus, we have shown that $F \neq G \Rightarrow F \not\sim G$ or that $F \sim G \Longrightarrow F \rightarrowtail G$. Hence, $\simeq$ contains any equivalence relation that obeys (a)-(d).

QED

C. The Minimality of Fixed-Points Produced by Y.

In this section we shall show that YF is, in a sense, the worst solution for A in the equation A=FA. We assume throughout that only α and β conversions are permissable; i.e., A=B means A and B are interconvertible by rules α and β and $A \supset B$ means either E[B] does not have a β-normal form or E[A]=E[B], for any E[ ]. Apparently, the results can be extended to a system permitting δ-reductions; but we shall not undertake to do so. The failure of Lemma 1′ in Chapter II for a system permitting η-conversions prevents extension of the results to such systems. Nevertheless, it seems likely that the results remain true for a full β-η-δ-calculus; and we regard the theorems in this section as settling the most difficult parts of the matter.

Definition: Let R be a set of disjoint subexpressions of a wfe, A. We say that A matches B except at R if A can be transformed into B by replacement of only subexpressions in R.

Example 1. Consider the two wfes

$$A \equiv (\lambda a.(\underline{\lambda z.a})a)(b\underline{c})\underline{d}$$

and $$B \equiv (\lambda a.(xa)a)(be)f$$

A matches B except at the three underlined subexpressions; for replacement of $(\lambda z.a)$ by (xa), c by e, and d by f transforms A into B.

Lemma 1. Suppose A matches B except at R and A' results from a β-contraction of a redex of A such that

(1) the redex is not a part of a member of R and

(2) the rator of the redex is not a member of R.

Then there exists a B' such that a single β-contraction on B yields B' and A' matches B' except at R' where R' is contained in the set of sons of members of R.

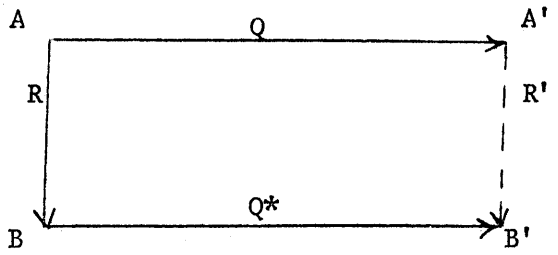
Example 1. (continued)

Since R consists of the three underlined subexpressions of A, requirements (1) and (2) are true of the redex $(\lambda a.(\underline{\lambda z.a})a)(b\underline{c})$; but the redex $(\underline{\lambda z.a})a$ fails requirement (2).

Proof (of Lemma 1):

Let $(\lambda x.M)N$ be the redex in A. By hypothesis it is not a part of a member of R, nor is $(\lambda x.M)$ a member of R. Therefore, the replacements that carry A into B must leave an homologous subexpression $(\lambda x.M^*)N^*$ in B where M* and N* result from certain replacements in M and N, respectively. Let B' be the result of contracting that redex.

The relations between A, A', B and B' are depicted below.



R stands for the replacements which carry A into B. Q stands for the replacement of $(\lambda x.M)N$ by subst[N,x,M]. Q* stands for the replacement of $(\lambda x.M^*)N^*$ by subst[N*,x,M*]. The operations denoted by R' remain to be derived.

Let us subdivide the members of R into three groups.

R_1 = subexpressions disjoint from the redex $(\lambda x.M)N$

R_2 = parts of M

R_3 = parts of N

By the definition of the father relation, the sons of R_1 in A' are homologous to their fathers in A and disjoint from the contractum of $(\lambda x.M)N$. Therefore, we can transform A' into B' by replacing the sons of R_1 by whatever they were replaced with before, and replacing the contractum of $(\lambda x.M)N$ by the contractum of $(\lambda x.M^*)N^*$. Thus, to prove the Lemma, we need only show that subst[N^*, x,M*] can be derived from subst[N,x,M] by replacements of sons of R_2 and R_3. The set R' will then consist of the sons R_1 and certain sons of R_2 and R_3.

Example 1 (continued).

Consider

$$A \equiv (\lambda a.(\underset{2}{\lambda z.a})a)(\underset{3}{bc})\underset{1}{d}$$

$$A' \equiv (\underset{2}{\lambda z.(bc)})(\underset{3}{bc})\underset{1}{d}$$

$$B' \equiv (x(be))(be)f$$

We have designated the members of R_1, R_2 and R_3 in A and their sons in A' by appropriate numbering of the underlinings. Clearly A' is transformed into B' by replacement of d by f and $\underline{(\lambda z.(bc))}(b\underline{c})$ by (x(be))(be).

Let us consider how subst[N*,x,M*] might be derived from M in two steps.

1a. Replace all the members of R_2 in M to derive M*.

2a. Replace the free occurrences of x in M* by N*.

Now consider the following alternative operations on M.

1b. Replace the free occurrences of x in M by N, yielding subst[N,x,M].

2b. Replace the sons of R_2 in M by the homologous expressions in subst[N*,x,M*]

3b. Consider the remaining sons of R_3 in the occurrences of N which were not displaced by Step 2b. Replace those so as to transform each occurrence of N into an occurrence of N*.

Example 1. (concluded)

Step 1a. carries $(\lambda z.a)a$ into $(xa)a$

Step 2a. carries this into $(x(be))(be)$

Step 1b. carries $(\underline{\lambda z.a})a$ into $\underline{(\lambda z.(bc))}(b\underline{c})$

Step 2b. carries this into $(x(be))(b\underline{c})$

Step 3b. carries this into $(x(be))(be)$

We argue that this second sequence of operations derives the same expression from M as the first. Hence, since the result of Step 2a is subst[N*,x,M*] and the result of Step 1b is subst[N,x,M], Steps 2b and 3b carry subst[N,x,M] into subst[N*,x,M*]. Notice that the replacements are of disjoint subexpressions. Thus, these subexpressions, together with the sons of R_1, constitute R' and are all sons of R.

QED

The purpose of this Lemma is to allow us to carry out parallel reductions of expressions which differ from each other in parts that are never vital to a given β-reduction.

Example 2

Let $A \equiv (\lambda x.xx)(\lambda x.xx)$

and $B \equiv Y(\lambda a.\lambda b.a)$

It is clear that $A \neq B$ for Ax is convertible only to Ax while Bx is convertible to B. On the other hand, $A \simeq B$. We shall show that $B \supset A$; the proof that $A \supset B$ follows from a similar argument.

Suppose for some E[ ], E[A] has a normal form. Then there is a standard reduction

$$E[A] \equiv E_0 > E_1 > \ldots > E_n$$

where E_n is in normal form. We shall construct a reduction of E[B]

$$E[B] \equiv F_0 > F_1 > \ldots > F_n$$

so that for each i the following is true: E_i matches F_i except at the

descendants of A (where F_i has a descendant of B instead).

This condition is obviously true for i=0.

Suppose the condition has been shown for some i<n. If the left-most redex in E_i is a descendant of A, it is of the form $(\lambda x.xx)(\lambda x.xx)$.

Then E_i cannot have a normal form, for contraction of that redex yields $(\lambda x.xx)(\lambda x.xx)$ again which must be the left-most redex in E_{i+1}. Since A contains no proper subexpressions which are redexes and cannot be a rator of a redex, the left-most redex in E_i must fulfill the conditions of Lemma 1. Therefore, choose F_{i+1} by Lemma 1 so that F_{i+1} matches E_{i+1} except at the descendants of A.

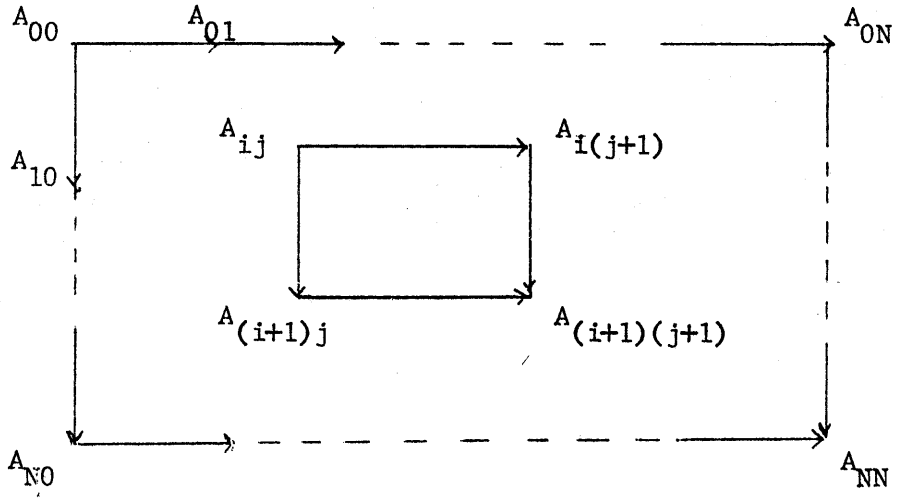
Now since E_n contains no redexes, and every descendant of A is a redex, E_n can contain no descendants of A (they must have been cancelled). Thus, since F_n can be derived from E_n by replacement of descendants of A, we must have $F_n \equiv E_n$.

Thus, since E[ ] was chosen arbitrarily, $B \supset A$.

The following theorem relies upon the Lemma of parallel moves stated in Chapter II (Lemma 1). It provides the basis for the main result of this section.

Theorem 6. Let $M \equiv (\lambda h.F(hh))(\lambda h.F(hh))$ for any wfe, F. For any context E[ ], if E[M] has a β-normal form, then there exists a number N such that $E[F^N(M)]$ can be reduced to β-normal form without the contraction of a part of a descendant of the occurrence of M.

Outline of Proof: The proof of this theorem is long and unpleasant. It is most simply visualized as the construction of a matrix of wfes as depicted below.



The sequence of wfes across the top $(A_{00},\ldots,A_{0N})$ constitutes the given standard order reduction of E[M] to β-normal form. The sequence down the left-hand side $(A_{00},\ldots,A_{N0})$ constitutes a reduction of E[M] to $E[F^N(M)]$. The sequence across the bottom $(A_{N0},\ldots,A_{NN})$ is the required reduction of $E[F^N(M)]$ to normal form, in which no

descendant of M is contracted. The sequence of expressions $A_{0N},\ldots,A_{NN}$ on the right hand side are all the same (i.e., $A_{0N} \equiv A_{1N} \equiv \ldots \equiv A_{NN}$) and in normal form.

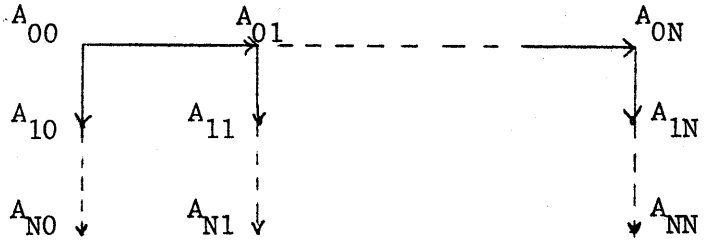
We shall define all the A_{ij} by specifying how each one is derived from the one above it ($A_{(i-1)j}$). Then we show that each A_{ij} can be reduced to its neighbor on the right ($A_{i(j+1)}$), thus completing each small rectangle.

Proof:

Consider the standard order reduction of E[M]:

$$E[M] \equiv A_{00} > A_{01} > \ldots > A_{0N}$$

where A_{0N} is in β-normal form. We shall define an (N+1) by (N+1) matrix of expressions as depicted below



where $A_{ij} \geq A_{(i+1)j}$.

Associated with each A_{ij} will be N disjoint sets of redexes $R^0_{ij},\ldots, R^N_{ij}$ which are parts of the expression A_{ij}. We define the A_{ij} and the R^k_{ij} inductively as follows:

1. $A_{00} \equiv E[M]$ and R^0_{00} consists of just that occurrence of M in E[M]. All other R^k_{00} are empty.
2. Given A_{0j} and $R^0_{0j},\ldots, R^N_{0j}$ suppose $A_{0(j+1)}$ is derived from A_{0j} by contraction of a redex Q_j.

We consider the following two cases:

Case 1. Q_j is not a member of R^p_{0j} for any p. Then $R^k_{0(j+1)} = \{$the sons of $R^k_{0j}\}$ for all k.

Case 2. Q_j is a member of R^p_{0j}. Then if $k \neq p+1$, $R^k_{0(j+1)} = \{$the sons of $R^k_{0j}\}$. To define $R^{p+1}_{0(j+1)}$, we note that Q_j must be a redex of the form $M' \equiv (\lambda h.F'(hh))(\lambda h.F'(hh))$ which differs from M only in that certain free variables of F have been replaced yielding F'. The contractum of Q_j is then F'(M'). Now $R^{p+1}_{0(j+1)}$ consists of that new occurrence of M' (which is the son of the first (hh) and therefore a member of no other $R^k_{0(j+1)}$), together with all the sons of R^{p+1}_{0j}.

Notice that R^k_{0j} must be empty if $k > j$ since $R^{k+1}_{0(j+1)} \neq \emptyset$ implies either $R^{k+1}_{0j} \neq \emptyset$ or $Q_j \in R^k_{0j}$.

3. Given A_{ij} and $R^0_{ij}, \ldots, R^N_{ij}$, we derive $A_{(i+1)j}$ by a complete reduction relative to the set of redexes R^i_{ij}. By Lemma 1 of Chapter II, this uniquely defines $A_{(i+1)j}$ despite the vagueness about the order in which this complete reduction is carried out.

The $R^k_{(i+1)j}$ are defined as follows:

(a) If $k \neq i+1$ then $R^k_{(i+1)j} = \{$ the descendants of $R^k_{ij}\}$. Notice that if $k \leq i$, $R^k_{(i+1)j}$ happens to be empty since the complete reduction at stage k eradicated R^k_{kj}.

(b) Now every contraction in the complete reduction of R^i_{ij} produces a contractum of the form F'(M') as in Case 2 above. Each of these new M's has descendants in $A_{(i+1)j}$ and none of them belong to any $R^k_{(i+1)j}$ (for $k \neq i+1$). $R^{i+1}_{(i+1)j}$ consists of all these descendants of new M's together with the descendants of R^{i+1}_{ij}.

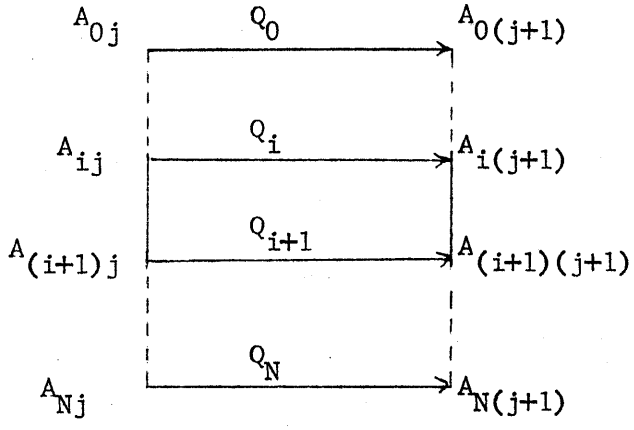
This completes the definition of the A_{ij} and R^k_{ij}. Roughly speaking, each member of R^k_{ij} is a pseudo-descendant of the original occurrence of M in A_{00}. It has arisen from k successive contractions of the form $M' > F'(M')$.

There are several degenerate cases which follow immediately from the definition.

1. $A_{i0} \equiv E[F^i(M)]$ for $0 \leq i \leq N$; and R^k_{i0} is empty unless i=k and in that case contains one member, the sole occurrence of M in A_{i0}.
2. Since A_{0N} has no redexes, R^k_{0N} is empty for all k. Hence, $A_{0N} \equiv A_{1N} \equiv \ldots \equiv A_{NN}$ and R^k_{iN} is empty for all i, k.
3. R^k_{0j} is empty if $j < k$. This is vacuously true for k=0; and, for $k > 0$, it follows from the definition by an inductive argument. Thus, we were justified in limiting the superscripts of the R's to being less than or equal to N.

To prove the theorem, we need only demonstrate, for j=0,...,N-1, that either $A_{Nj} \equiv A_{N(j+1)}$ or $A_{Nj} > A_{N(j+1)}$ via a contraction of a redex which is not a part of any redex in R^N_{Nj}.

Our strategy shall be to construct the "ladder" depicted below for any j.



Each "rung" of the "ladder", Q_i, is either null in the case that $A_{ij} \equiv A_{i(j+1)}$ (in which case, all "rungs" below it must be null), or it is a reduction of a redex, Q_i, in A_{ij}.

Case 1: Suppose Q_0 is a member of R^p_{0j} for some p.

Lemma A. For all $0 \leq i \leq p$ there exists Q_i such that

(1) Q_i is a member of R^p_{ij} and is the left-most redex in A_{ij}.

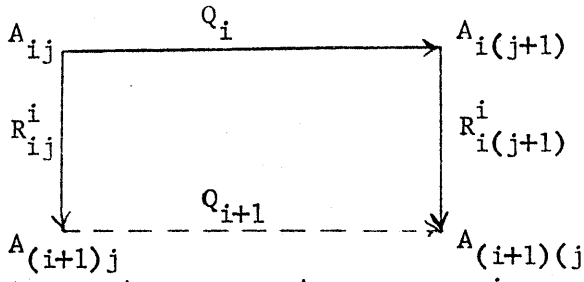
(2) Contraction of Q_i carries A_{ij} into $A_{i(j+1)}$

(3) For all $0 \leq k \leq N$, $R^k_{i(j+1)}$ consists of just the sons of R^k_{ij} with respect to the contraction of Q_i.

Proof: By induction. For i=0 (1) and (2) are true by assumption and (3) is true by definition of $R^k_{0(j+1)}$.

Suppose the conditions are true for $i < p$.

Pictorially we have



By (3) the passage from A_{ij} to $A_{i(j+1)}$ to $A_{(i+1)(j+1)}$ is a complete reduction of A_{ij} relative to the set of redexes $\{Q_i\} \cup R^i_{ij}$. Since, by (1), Q_i is the left-most redex in A_{ij}, it has a unique descendant via the complete reduction relative to R^i_{ij}, which we call Q_{i+1}. It is a member of $R^p_{(i+1)j}$ which establishes (1) for i+1. Then the complete reduction relative to R^i_{ij} followed by a contraction of Q_{i+1} is again a complete reduction relative to $\{Q_i\}\ R^i_{ij}$. Therefore, by Lemma 1 of Chapter 2, the result is $A_{(i+1)\ (j+1)}$ which establishes (2) for i+1. By (3) for i, the $R^k_{i(j+1)}$ are the descendants of R^k_{ij} for all k. By Lemma 2 every subexpression in $A_{(i+1)(j+1)}$ has the same ancestor in A_{ij} regardless of which reduction sequence is chosen. Thus, each member of $R^k_{(i+1)(j+1)}$ as determined by the reduction via Q_i is also a descendant of a redex in R^k_{ij} via the reduction ending with a contraction of Q_{i+1}. More specifically, each member of $R^k_{(i+1)(j+1)}$ is a son of a member of $R^k_{(i+1)j}$. Thus, we have shown (3) for i+1.

QED

Therefore, the Lemma is proved, specifically for i=p. Now we assert that $A_{(p+1)j} \equiv A_{(p+1)(j+1)}$. For the contraction of Q_p followed by a complete reduction of $R^p_{p(j+1)}$ is, by the foregoing Lemma, a complete reduction of A_{pj} relative to R^p_{pj}. But the passage from A_{pj} to $A_{(p+1)j}$ is, by definition, a complete reduction relative to R^p_{pj}.

Hence, we have $A_{ij} \equiv A_{i(j+1)}$ for i=p+1,...,N. In particular, we have

$$A_{Nj} \equiv A_{N(j+1)}$$

Case 2. Suppose Q_0 is not a member of R^p_{0j} for any p.

Lemma B. For all $0 \leq i \leq N$ there exists a Q_i such that

(1) Q_i is not part of a member of any R^p_{ij}, and is the left-most redex in A_{ij}.

(2) Contraction of Q_i carries A_{ij} into $A_{i(j+1)}$

(3) For all $0 \leq k \leq N$, $R^k_{i(j+1)}$ consists of just the sons of R^k_{ij} with respect to the contraction of Q_i.

Proof: By induction. Since the reduction of A_{00} to A_{0N} is standard Q_0 can be a part of a member of R^p_{ij} only if it is a member itself since every member of R^p_{ij} is a redex. This establishes (1); (2) and (3) are obviously true.

Suppose the conditions are true for $i < N$. By the arguments of Lemma A, there is a unique descendant of Q_i in $A_{(i+1)j}$, which we call Q_{i+1}. Further, since Q_i is to the left of all the redexes in A_{ij}, Q_{i+1} must be to the left of all the redexes in $A_{(i+1)j}$. In particular, it is to the left of all the redexes in $R^k_{(i+1)j}$ and thus cannot be a part of any of them. This establishes (1) for i+1.

(2) and (3) can be established by the arguments used in Lemma A.

QED

In the final case i=N, we have shown that $A_{Nj} > A_{N(j+1)}$ by contraction of a redex which is not a part of any member of R^N_{ij} which is the set of descendants of M in $A_{N0} \equiv E[F^N(M)]$.

Since we have shown this for all j, we have a sequence:

$$E[F^N(M)] \equiv A_{N0} \geq A_{N1} \geq \ldots \geq A_{NN} \equiv A_{0N}$$

in which no part of a descendant of M was contracted

QED

Theorem 7. If $A \supset FA$ then $A \supset YF$.

Proof: Suppose for any E[ ], E[YF] has a β-normal form. Then by Theorem 6, there is an N such that $E[F^N(M)]$ (where $M \equiv (\lambda h.F(hh))(\lambda h.F(hh)) < YF$) can be reduced to normal form without a contraction of a part of a descendant of M. Let the reduction be

$$E[F^N(M)] \equiv B_0 > B_1 > \ldots > B_k$$

when B_k is in normal form. We shall construct a reduction

$$E[F^N(A)] \equiv C_0 > C_1 > C_k$$

such that B_i matches C_i except at the descendants of M (where C_i has a descendant of A).

Clearly, the condition is true for i=0. Suppose it holds from some $i < k$. Since the passage from B_i to B_{i+1} does not involve a contraction internal to a descendant of M, and since no descendant of M can be the rator of a redex, Lemma 1 applies. We can choose $C_{i+1} < C_i$ by Lemma 1 so that B_{i+1} matches C_{i+1} except at the descendants of M.

Thus, by finite induction, B_k matches C_k except at the descendants of M. But since B_k is in normal form, it may contain no descendants of M since they must be redexes. Therefore, $C_k \equiv B_k$.

Since E[ ] was chosen arbitrarily, we have shown $F^N(A) \supset F^N(M)$. Further, we have $YF > M > F(M) \geq F^2(M) > \ldots > F^N(M)$ and also $A \supset F(A) \supset F^2(A) \supset \ldots F^N(A)$ by the monotonicity of $\supset$. Therefore, $A \supset F^N(A) \supset F^N(M) = YF$.

QED

Corollary 7(a). If A = FA then $A \supset YF$.

Proof. Immediate since A=FA implies $A \supset FA$.

Corollary 7(b). If A=FA then $\phi_n(A)$ is a functional extension of $\phi_n(YF)$ for any n.

Proof. Immediate from Corollary 7(a) and Theorem 3.

Remarks:

It is easy to see, intuitively, that A must be an extension of YF if A is convertible to FA. In a sense, YF must be converted to F(YF) if it is to serve a purpose in any larger reduction while A may be converted to FA but may also reduce in some other way. Thus, the would-be reducer of an expression has more options available if A is used instead of YF.

Example 3. Consider the following PAL [20] definitions

$$F(x)(y) = H(x)$$

$$\underline{\text{where}}\ \underline{\text{rec}}\ H(z) = pzy(H(mz))$$

and $$\underline{\text{rec}}\ G(a)\ (b) = pab(G(ma)b)$$

It should be clear that F and G denote the same function; the only difference seems to be that G passes the argument b at each recursion while F does not pass y. Recasting these definitions as simple wfes, we have

$$F \equiv \lambda x.\lambda y.YHx$$

$$H \equiv \lambda f.\lambda z.pzy(f(mz))$$

$$G \equiv YK$$

$$K \equiv \lambda g.\lambda a.\lambda b.pab(g(ma)b)$$

We shall show that $F \simeq G$. First, we show $F \supset G$. By Corollary 7(a) we need only show that $K(F) = F$. This is easily done by a sequence of contractions and expansions of $K(F)$.

$$\begin{aligned} K(F) &> \lambda a.\lambda b.pab((\lambda x.\lambda y.YHx)(ma)b) \\ &> \lambda a.\lambda b.pab(YH'(ma)) \\ &\quad (\text{where } H' \equiv subst[b,y,H]) \\ &< \lambda a.\lambda b.(\lambda z.pzb(YH'(mz)))a \\ &< \lambda a.\lambda b.YH'a \\ &=_{\alpha} \lambda x.\lambda y.YHx \\ &\equiv F \end{aligned}$$

Thus, we have $K(F) = F$, hence $F \supset YK \equiv G$

To show that $G \supset F$, notice that

$$G > \lambda a.\lambda b.pab(YK(ma)b)$$

$$< \lambda a.\lambda b.pab((\lambda z.YKzb)(ma))$$

and $$F > \lambda x.\lambda y.pxy(\quad YH \quad (mx))$$

$$=_{\alpha} \lambda a.\lambda b.pab(\quad YH' \quad (ma))$$

Thus, by the monotonicity of $\supset$, we need only show that

$$(\lambda z.YKzb) \supset YH'$$

Consider $H'(\lambda x.YKzb) > \lambda z.pzb((\lambda z.YKzb)mz))$

$$> \lambda z.pzb(YK(mz)b)$$

and $(\lambda x.YKzb) > \lambda z.K(YK)zb$

$$> \lambda z.(\lambda a.\lambda b.pab(YK(ma)b))zb$$

$$> \lambda z.pzb(YK(mz)b)$$

Thus, $H'(\lambda z.YKzb) = (\lambda z.YKzb)$ and by Corollary 7a.

$$(\lambda z.YKzy) \supset YH'$$

Example 4

For any wfe, A, $A=(\lambda x.x)A$, hence $A \supset Y(\lambda x.x)$. Notice, however, that $Y(\lambda x.x) = (\lambda h.hh)(\lambda h.hh)$ which is equivalent to the null function.

Bohm [15] has given a technique by which we may generate an infinite number of (apparently) non-interconvertible versions of Y. Each Y_i has the property that

$$Y_i F = F(Y_i F) \text{ for any wfe, } F.$$

We define, Y_i, inductively as follows:

$$Y_0 \equiv Y \equiv \lambda f.(\lambda h.f(hh))(\lambda h.f(hh))$$

and $$Y_{i+1} \equiv Y_i G$$

where $G \equiv \lambda y.\lambda f.f(yf)$

Theorem 8. For all i, $Y_i = GY_i$

Proof: For i=0

$$Y_0 > \lambda f.f((\lambda h.f(hh))(\lambda h.f(hh)))$$
$$< \lambda f.f(Y_0 f)$$
$$< GY_0$$

Suppose we have $Y_i = GY_i$ for some i. Then

$$Y_{i+1} \equiv Y_i G = (GY_i)G \equiv (\lambda y.\lambda f.f(yf))Y_i G$$
$$> G(Y_i G) \equiv GY_{i+1}$$

Thus, by induction, it is established for all i

QED

Corollary 8 (a). For all i, and all wfes, F,

$$Y_i F = F(Y_i F)$$

Proof: Immediate from Theorem 8.

Example 5.

$$Y_1 = (\lambda h.\lambda f.f(hhf))(\lambda h.\lambda f.f(hhf))$$

$$> (\lambda f.f(Y_1 f))$$

$$Y_2 \equiv Y_1 G > (\lambda f.f(Y_1 f))G$$

$$> G(Y_1 G)$$

$$> \lambda f.f(Y_1 G f)$$

Böhm has shown that $Y_0 \neq Y_1$. It seems likely that $Y_i \neq Y_j$ for $i \neq j$.

Are all these Y_i's equivalent ($\simeq$)? Does Theorem 7 still hold if we substitute some Y_i for Y in its statement? We give a partial answer to these questions with the following:

Corollary 8(b). For all $i \geq 0$

(1) for any wfes, A and F: if $A \supset FA$ then $A \supset Y_i F$.

(2) $Y_i \supset Y_{i+1}$

(3) If $i > 0$ then $Y_{i+1} \supset Y_i$.

Proof: First notice that, for any i, (1) $\Rightarrow$ (2). For by Theorem 8, $Y_i = GY_i$. Substituting Y_i for A and G for F in (1) we arrive at $Y_i \supset GY_i \Rightarrow Y_i \supset Y_i G \equiv Y_{i+1}$. Hence $Y_i \supset Y_{i+1}$.

Now we prove the corollary by induction on i.

for i=0, (1) is simply Theorem 7 and (3) is vacuously true.

Suppose (1), (2) and (3) hold for all $i \leq k$. $Y_k \supset Y_{k+1}$ implies $Y_k F \supset Y_{k+1} F$, for any F, by the monotonicity of $\supset$. Thus, for any A, $A \supset FA \Rightarrow A \supset Y_k F \supset Y_{k+1} F$. Thus, (1) and (2) are established for k+1.

To show (3), notice that $Y_{k+2} = G(Y_{k+2})$ implies $Y_{k+2} \supset Y_k G \equiv Y_{k+1}$, by (1) for k. Thus, (1), (2) and (3) are established for all $i \leq k+1$.

QED

This corollary shows that Theorem 7 is true for any Y_i and that $Y_0 \supset Y_1 \simeq Y_2 \simeq \cdots \simeq Y_i \cdots$. It is probably true that $Y_1 \supset Y_0$, but it is not so easily proved.

Example 6. Recursion Induction

Suppose we have two wfes, A and B, which we wish to prove equivalent in a certain class of contexts. One way of doing this is to show that

(1) A = FA

(2) B = FB

and (3) E[YF] has a normal form

for certain contexts, E[ ].

From Theorem 7, it follows that $A \supset YF$ and $B \supset YF$. Thus, if E[YF] has a normal form, E[A] = E[YF] = E[B], and A is equivalent to B in those contexts.

This technique is quite reminiscent of McCarthy's Principle of Recursion Induction [17]. To show the similarity, we carry out a typical proof by recursion induction, found in [17], justifying most of the steps by λ-calculus conversions. This has the effect of lengthening the proof considerably.

The problem is, given the following recursive definition:

$m+n$ = **if** $n=0$ **then** m **else** $m' + n^-$

(where $m'=m+1$ and $n^- = n-1$)

prove that

$(m+n)' = m'+n$

We translate this problem into a more formal, λ-calculus, one as follows:

(1) Introduce the constants s and p for the successor and predecessor functions; i.e., (sx) means x' and (px) means x^-

(2) Define

$A \equiv YP$

where $P \equiv \lambda a.\lambda m.\lambda n.$ **if** $n=0$ **then** m **else** $a\ (sm)(pn)$

(3) Prove $C \simeq B$

where $C \equiv (\lambda m.\lambda n.s(A\ mn))$ and $B \equiv (\lambda m.\lambda n.A(sm)n)$ for any context E[ ] where m and n are bound only to numerals.

Now let $F \equiv \lambda f.\lambda m.\lambda n.$ **if** $n=0$ **then** (sm) **else** $f(sm)(pn)$

$C \equiv \lambda m.\lambda n.s(A\ mn)$

$> \lambda m.\lambda n.s($**if** $n=0$ **then** m **else** $A\ (sm)(pn))$

$\supset \lambda m.\lambda n.$ **if** $n=0$ **then** sm **else** $s(A\ (sm)(pn))$

$< \lambda m.\lambda n.$ **if** $n=0$ **then** sm **else** $(\lambda m.\lambda n.s(A\ mn))\ (sm)\ (pn)$

$< F(\lambda m.\lambda n.\ (PLUS\ mn)) \equiv FC$

Thus, we have $C \supset FC$; hence $C \supset YF$.

$B \equiv \lambda m.\lambda n.\ A\ (sm)n$

$> \lambda m.\lambda n.$ **if** $n=0$ **then** sm **else** $A\ (s(sm))(pn)$

$< \lambda m.\lambda n.$ **if** $n=0$ **then** sm **else** $(\lambda m.\lambda n.\ A\ (sm)n)(sm)(pn)$

$< F(\lambda m.\lambda n.\ A\ (sm)n) \quad FB$

Thus, we have $B = FB$, hence $B \supset YF$.

Assuming that $YFxy$ has a normal form, if x and y are numerals, we have

$Cxy = Bxy$ for any numerals x and y.

Notice that the meaning of s and p never entered into the proof (this is not always the case).

D. Conclusions

1. Summary

a. Kleene's first recursion theorem provides a simple, semantic characterization of the function defined by a recursive equation.

b. Kleene's theorem is not true if the computation procedure is changed to allow unrestricted substitutes analogous to β-conversion. The functions computed by the alternate rules may be extensions of the functions computed by the original rules.

c. The notion of one wfe being an extension of another ($\supset$) is derived from the assumption that a wfe's not having a normal form is analogous to a computation's not terminating.

d. Extensional equivalence ($\simeq$) is defined and its properties studied.

e. The main result of the chapter is an analogue to Kleene's theorem for the λ-calculus: For any wfe, F, if A=FA, then $A \supset YF$.

d. The result is shown to be invariant over many equivalent wfes, Y_i, each of which has the property that $Y_iF = F(Y_iF)$ for all F.

2. Support of Thesis

If one interprets a wfe, F, as a functional, then YF may be interpreted as that fixed-point of F which is extended by every other fixed-point of F. We argue that this is a semantic interpretation of recursion as opposed to the purely mechanical one depending upon the definition of Y.

The fact that many different versions of Y are equivalent but not interconvertible makes the semantic characterization more attractive than one based on any particular version of Y.

Further discussion of the thesis appears in Chapter V.

3. Order of Evaluation

The correspondence between our theorem and the Kleene recursion theorem is complicated by the fact that a wfe may be reduced in any order while Kleene's system of formal computation restricts substitutions. The difference between the theorems can be related to ALGOL-60: Kleene's result applies to recursive functions in which all parameters are given by value while ours applies to cases in which call by name is used. The example of

Section A, Part 3, may be translated into ALGOL to illustrate the point.

```
integer procedure f(m,n); value m,n;
if m=0 then f:=0 else f:= f(m-1,f(n-2,m))
```

With this definition, f(m,n), is defined only if m=0 or m=1 and n=2. If we erase "value m,n" then f(m,n) is defined whenever $m \geq 0$.

While the unrestricted order of reduction is an interesting property of the λ-calculus, we believe that a realistic programming language based on the λ-calculus should use a restricted reduction order analogous to call by value. The algorithm of Landin's SECD machine [13] or LISP's eval are examples of restricted reduction rules of this nature.

Aside from purely practical reasons for this preference, we cite the more complicated definition of extension in Theorem 2 (p. 48) which was necessary to characterize the minimal nature of the fixed-point solution produced by Y in a system with unrestricted reduction rules.

CHAPTER IV

Types

A. Motivation

A typical programming language (e.g., ALGOL or FORTRAN) has many different kinds of things in its universe of values. For example, an ALGOL variable may denote a number, string, function, label or switch. A programmer is usually required to declare which type of object each variable in his program may assume.

In general, the type system of a programming language calls for a partitioning of the universe of values presumed for the language. Each subset of this partition is called a type.

From a purely formal viewpoint, types constitute something of a complication. One would feel freer with a system in which there was only one type of object. Certain subclasses of the universe may have distinctive properties, but that does not necessiate an a priori classification into types. If types have no official status in a programming language, the user need not bother with declarations or type checking. To be sure, he must know what sorts of objects he is talking about, but it is unlikely that their critical properties can be summarized by a simple type system (e.g., prime numbers, ordered lists of numbers, ages, dates, etc.).

Nevertheless, there are good, pragmatic reasons for including a type system in the specifications of a language. The basic fact is that people _believe_ in types. A number is a different kind of thing from a pair of numbers; notwithstanding the fact that pairs can be represented by numbers. It is unlikely that we would be interested in the second component of 3 or the square root of $< 2,5 >$. Given such predispositions of human language users, it behooves the language designer to incorporate distinctions between types into his language. Doing so permits an implementer of the language to choose different representations for different types of objects, taking advantage of the limited contexts in which they will be used.

Even though a type system is presumably derived from the natural prejudices of a general user community, there is no guarantee that the tenets of the type system will be natural to individual programmers. Therefore it is important that the type restrictions be simple to explain and learn. Furthermore, it is helpful if the processors of the language detect and report on violations of the type restrictions in programs submitted to them. This activity is called _type-checking_.

B. Perspective

It is clear that the kind of undefinedness associated with nonterminating computations cannot be prevented if the language in question is a universal one. Our only aim is to provide for the undefinedness that arises from so-called don't-care conditions in language specifications.

For purposes of discussion, let us assume that a language has been specified by

(1) a partial mapping from character strings to formal expressions (as in Ch. 2 A); and

(2) a program that interprets formal expressions.

We shall not concern ourselves with the details of (1); we shall simply assume that it filters out certain "illegal" character strings and produces a tree structure. The interpreter (2) accepts a parse tree and performs certain manipulations on it and may eventually halt yielding a final tree, the answer. At certain points in the interpreter, don't care conditions are indicated by conditional expressions involving the special word ERROR. The intent is that any tree that causes the interpreter to reach ERROR is illegal in some sense.

Example 1. LISP [9]

LISP is a language which does not happen to require phase (1) in any significant way. Phase (2) is embodied in the functions apply and eval. Simple changes in the definitions of these functions will make the don't care conditions more explicit. For example, we may replace the phrase

$$eq[fn; CAR] \rightarrow caar[x]$$

in apply by

eq[fn; CAR] → [atom [car[x]] → ERROR; T→caar[x]].

This makes explicit the fact that the car of an atom is undefined.

Naturally, a specification of this kind is easily transformed into an implementation, albeit inefficient. We need only choose a representation for parse trees and write the interpreter program. The interpreter would be expected to render suitable diagnostics when an ERROR was reached. We often say that such an implementation uses dynamic type checking; each object would include an indication of its type, and each application of a primitive function would follow a check to insure that the arguments have proper type.

A more efficient implementation calls for further processing of the tree to produce a representation that can be interpreted more easily, preferably by the hardware of the target computer. We are not concerned here with code generation problems but only with the treatment of the don't care situations. In the cause of efficiency a natural inclination is to take advantage of them by not checking types at run time and letting the bits fall where they may, so to speak, if don't care situations arise. To do such while maintaining the integrity of the implementation requires that the preprocessing of the parse tree filter out and report on trees which, when evaluated, will end in an ERROR. We call this activity static type checking.

Once committed to any significant preprocessing of the tree, we shall require the ability to preprocess, or compile, different parts of an expression independently, for the overhead associated with the compiling prohibits recompiling large expressions after making small changes. We have been implicitly unifying the notion of program and data; an expression may be thought of as program or data.

In the expression (FX) (F applied to X) the expression F may be considered a program and X data. Thus, our notion of independent compilation includes also the standard situation in which a program, F, may be compiled and then applied to many different data sets, X, which may have been compiled themselves. If we consider FORTRAN in this light, we note that it does not provide a facility for compiling its data independently, but, rather, compiles it at run-time with input routines. Naturally, independent compilation also includes the meshing of different programs.

The need for independent compilation places further requirements on any proposed scheme for type-checking. It will be necessary to insure that any individual expression submitted for compilation be free from ERRORs, and when two compiled expressions are combined in any way, the operating system must insure that their interaction will not promote ERRORs.

The general approach we shall take runs roughly as follows:

(a) When an expression is submitted for compilation, it is checked for type errors. If it passes, the output consists of the compiled representation together with a certain amount of type information.

(b) When two compiled expressions are combined, the operating system checks their type information to insure that they can be combined and, if so, derives some new type information to be associated with the combined whole.

The major subject of this chapter is the design of this type information component.

The main emphasis of our work is on functional types, i.e., types of function variables. While this may appear strange, given the rather sparing use of function variables by most programmers, we argue that it is a reasonable approach.

First, function variables do appear in many languages, (ALGOL, FORTRAN, MAD, LISP) and the type checking problems they introduce must be solved if complete checking is to be possible.

Second, functional types can be assigned to primitive operators such as '+', as well as programmer-defined functions or function variables. This unification allows us to handle the usual type-checking problems associated with primitive operators in a consistent way. After all, aside from syntax, a primitive operator symbol is just a function variable whose value is fixed for the language.

Third, some of the problems associated with independently compiled functions gain perspicuity when looked upon as function variable problems. For example, if the function ($\lambda x.A$) refers to another function B which is to be defined and compiled elsewhere

the loading process can be assumed to carry out the application of (λB.λx.A) to B in some way or another.

Finally, we feel that the functional is an elegant notion but that its use is not well understood or appreciated. An attempt to reconcile functionals with the requirement of type checking should be of interest independent of its practical significance.

C. The Basic System

We shall describe a very simple, primitive language - a λ-δ-calculus - which has certain don't care conditions. Then we shall introduce a system of types to allow a preprocessor to insure that a given expression will not give rise to a don't care condition.

1. The Object Language

We shall consider a restricted λ-calculus in which only β-reductions and a certain type of δ-reductions are permitted. The rule of β-reduction is as described in Chapter II, section B (i.e., $(\lambda x.M)N$ may be replaced by subst[N,x,M]). Each δ-rule has the following form:

(C_1C_2) may be replaced by C_3

where C_1, C_2 and C_3 are specific constants.

Furthermore, we define the father of C_3 in the resulting expression to be the combination (C_1C_2) that occurred in the original expression. Recall that the father function was defined for β-reduction in Chapter II, section C.

For the sake of discussion, let us choose a particular set of constants and rules to carry out arithmetic. There will be an infinite number of constants and rules, but it will be clear that evaluation is an effective process. There are three classes of constants.

Numerals: $0,1,2,\ldots,-1,-2,-3,\ldots$

Unary Functions: add_0, add_1, add_2,..., add_{-1},...

sub_0, sub_1, sub_2,..., sub_{-1},...

Binary Functions: sum, difference

The rules are as follows:

if $\bar{n}$ is a numeral denoting the number n,

$(\text{add}_i \bar{n})$ may be replaced by $\overline{n+i}$

$(\text{sub}_i \bar{n})$ may be replaced by $\overline{n-i}$

$(\text{sum } \bar{n})$ may be replaced by add_n

$(\text{difference } \bar{n})$ may be replaced by sub_n

Thus, $((\text{sum } \bar{n})\bar{m}) = (\text{add}_n \bar{m}) = \overline{m + n}$

and $(\text{difference } \bar{n}\ \bar{m}) = (\text{sub}_n \bar{m}) = \overline{m - n}$

For the moment, let us assume that the computer for this language reduces expressions by the standard order rule (i.e., left-most redex first). The evaluator thus specified defines a partial function from the set of all wfes to those in normal form. However, assuming that the intended use of this language is to do arithmetic, there are many wfes which are not significant. For example (3 2) or $(\text{add}_0\ (\lambda x.x))$ would not be meaningful in any context; i.e., they are don't care expressions.

These don't care situations could be filled in by the following additional rule: Suppose C is a constant, and E is any expression in normal form with no free variables. If (C E) cannot be reduced by any other rules, then (C E) may be replaced by the new constant ERROR.

The type checking problem can then be phrased as follows: Given a wfe, insure that evaluation of it will not result in a wfe

containing ERROR. A second problem, the one we shall actually consider, is: Given a wfe, insure that no series of reductions, in any order, will result in a wfe containing ERROR. Obviously, this second condition guarantees the first, but not vice versa. For example, the evaluator will reduce $((\lambda x.5)(1\ 2))$ to 5 and halt while this expression is also reducible to $((\lambda x.5)\text{ERROR})$.

We argue that neither one of these problems is recursively decidable; i.e., there is no general algorithm for deciding whether a wfe has the required property. Suppose there was a procedure for deciding either of the questions; then we could use it to decide whether any wfe containing no free variables or constants had a normal form by the following stratagem:

Given A, a wfe with no constants or free variables, construct the wfe (C A) where C is one of the constants in the language (e.g., 0).

If (C A) is reducible to a wfe containing ERROR, the reduction that introduced ERROR must have been the replacement of (C A') by ERROR where A' is the normal form of A. This follows from the facts (1) that no reductions inside A can cause ERRORs since A contains no constants; and (2) (C X) is reducible only if X is in normal form. Thus, A must have a normal form.

Conversely, if A has a normal form then (CA) is reducible to ERROR if C is chosen so that it is applicable only to constants.

Thus, applying the alleged decision procedure to (C A) will yield an answer to the question of whether A has a normal form - a question which we know to be undecidable (see Corollary 4, Chapter II). Therefore, no such procedure can exist.

Although this argument depends upon the somewhat peculiar features of the system and its δ-rules, we claim that the same sort of result would hold for other systems of a similar kind. For example, the question of whether a LISP S-expression, when evaluated, results in an error (e.g., car applied to an atom) is also undecidable.

We shall now introduce a type system which, in effect, singles out a decidable subset of those wfes that are safe; i.e., cannot given rise to ERRORs. This will disqualify certain wfes which do not, in fact, cause ERRORS and thus reduce the expressive power of the language.

2. The Type System

The type system is inspired by Curry's theory of functionality [11] which he developed in relation to combinatory logic. The purpose of that work was to formulate a very simple logistic system as the basis for any formal logic and to explicate certain phenomena such as the paradoxes. Our more mundane goals will permit a considerably less rigorous development with emphasis on the more pragmatic features of the theory.

Syntax

A type expression (te) is either

an atomic type

or a functional type which has a domain which is a type expression

and a range which is a type expression

The written representation of type expressions is given by the following BNF productions.

< type expressions > ::= < primary > | < primary > → < type expression >

< primary > ::= < atomic type > | (< type expression >)

< atomic type > ::= t_1 t_2 t_3 ...

Semantics

We assume that every type expression denotes a subset of the universe of discourse and that these subsets are mutually exclusive and collectively exhaustive; i.e., that every object in the universe is in one, and only one, type. This is rather a large assumption, and we shall refer to it as The Stratification Hypothesis.

A particular type expression may be interpreted by the following rules:

1. The symbols t_1, t_2,..., denote sets of atomic objects (e.g., numbers, strings, etc.)
2. If α denotes the set A and β denotes the set B, then $\alpha \rightarrow \beta$ denotes the set of all functions (including partial functions) which map A into B. That is, $f \in \alpha \rightarrow \beta$ means f is a single-valued subset of $A \times B$.

We shall distinguish two kinds of undefinedness for the application of one object to another. Specifically, there are three possible results for f(x):

1. If, for some types α and β, $f \epsilon \alpha \rightarrow \beta$ and $x \epsilon \alpha$ and $\langle x,y \rangle \epsilon f$ for some y then f(x) = y; i.e., f(x) is <u>defined</u>
2. If, for some types α and β, $f \epsilon \alpha \rightarrow \beta$ and $x \epsilon \alpha$, but for no y is $\langle x,y \rangle \epsilon f$ then $f(x) = \omega$; i.e., f(x) is <u>ω-undefined</u>
3. If, for <u>no</u> types is it true that $f \epsilon \alpha \rightarrow \beta$ and $x \epsilon \alpha$ then f(x) = ERROR; i.e., f(x) is <u>E-undefined</u>.

Intuitively, ω-undefinedness is the intrinsic kind of undefinedness associated with non-terminating computations while E-undefinedness is associated with "type errors."

Let us particularize type expressions to deal with the arithmetic system described in the previous section. There is just one atomic type, N (for number); it follows that

Any numeral is contained in N

for all i, $\text{add}_i \epsilon N \rightarrow N$

for all i, $\text{sub}_i \epsilon N \rightarrow N$

sum $\epsilon N \rightarrow N \rightarrow N$ (i.e., $N \rightarrow (N \rightarrow N)$)

difference $\epsilon N \rightarrow N \rightarrow N$

In addition, we might say

(sum 3) $\epsilon N \rightarrow N$

$(\lambda x.\text{sum } xx)$ $\epsilon N \rightarrow N$

$(\lambda f.\ \text{add}_3(f\ 2))$ $\epsilon\ (N \rightarrow N) \rightarrow N$

On the other hand, (3 2) and (add_2 sub_1) are E-undefined and are not contained in any type.

In general, we assume that each constant in the language is assigned a fixed type. These permanent type assignments should reflect the reduction rules. Specifically, we require that (C_1 C_2) be reducible to C_3 only if C_1 $\epsilon \alpha \rightarrow \beta$, C_2 ϵ α and C_3 ϵ β, for some α and β. If these conditions are not fulfilled for C_1 and C_2 the combination (C_1 C_2) is reducible to ERROR.

Returning to the specific arithmetic constants, we see that those conditions are fulfilled. The following productions define the set of combinations, composed solely of constants, which the evaluator will process without encountering an ERROR.

< legal exprs > ::= < number > | < function > \ sum \ difference

< function > ::= (sum < number >) | (difference < number >) |
< add > | < sub >

< number > ::= (< function > < number >) | < numeral >

< add > ::= add_0 | add_1 | ... | add_{-1}

< sub > ::= sub_0 | sub_1 | ... | sub_{-1}

< numeral > ::= 0 | 1 | ... | -1

3. Type Assignments

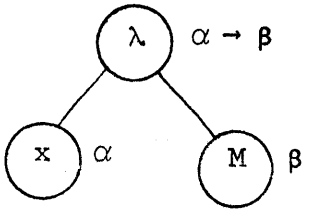
The type language and the object language are related through the notion of the type assignment, a minor generalization of the type declaration. Roughly speaking, the assignment of a type expression to a wfe of the object language is a declaration that the wfe will either be ω-undefined, or that its value will belong to the set denoted by the type expression.

A type assignment for a wfe consists of the association of a type with every subexpression of the wfe. The type assignment is valid if it obeys the following rules, which are motivated by the semantics discussed above:

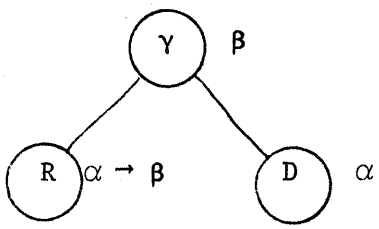
1. If S_1, S_2 are occurrences of the same variable, and both are free in the same subexpression, then they are assigned the same type.
2. If S is an occurrence of a λ-expression $(\lambda x.M)$ and x is assigned type α and M is assigned type β, then S is assigned type $\alpha \rightarrow \beta$. Thus, we give a λ-expression the natural interpretation as a function of its bound variable.
3. If S is an occurrence of a combination (R D) and is assigned type β, while D is assigned type α, then R is assigned type $\alpha \rightarrow \beta$.
4. A constant is always assigned its permanent type.

Rules 2 and 3 may be illustrated graphically.

Rule 2:



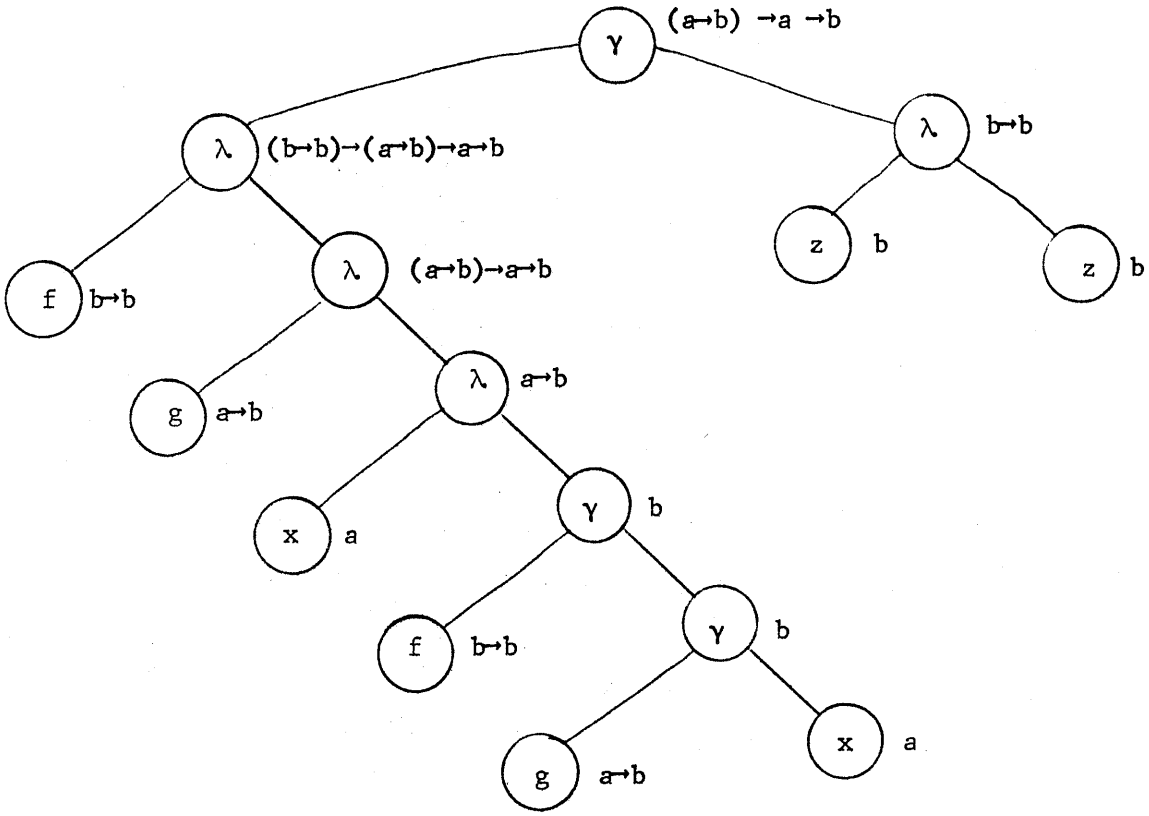
Rule 3:



In the sequel we shall take type assignment to mean valid type assignment.

Example 2.

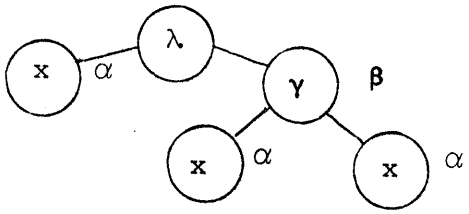
By writing the wfe $(\lambda f.\lambda g.\lambda x.f(gx))(\lambda z.z)$ in tree form we can illustrate a type assignment by writing the type expressions beside each node.



Inspection of the picture should reveal that replacement of a and b by any two type expressions will result in a valid type assignment (e.g., let $a \equiv t_1$ and $b \equiv t_2$).

Example 3.

Certain wfes admit to no type assignment even if they contain no constants. The wfe $(\lambda x.xx)$ cannot have a type assignment.



In the tree above it is clear that no type expressions α and β can satisfy Rule 2 for we must have

$$\alpha \equiv \alpha \rightarrow \beta$$

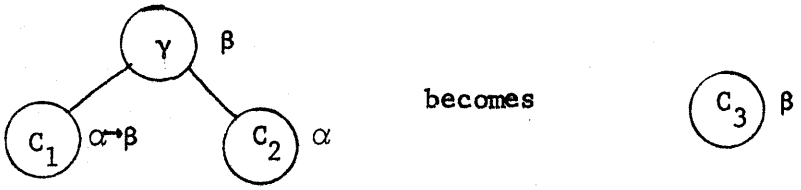
and no type expression can be equal to a subpart of itself. Notice that we use $\equiv$ for equality between type expressions since they are equal only if identical. This is a consequence of the stratification hypothesis.

D. Sufficiency of the Type System

In this section we shall show that the type system is sufficient in the sense that any wfe with a valid type assignment will not result in an ERROR when evaluated. This fact will follow from the demonstration that any wfe with a type assignment will, in a sense, bequeath a similar type assignment to any wfe it can be reduced to by the evaluator rules. Thus, every stage in an evaluation will involve a legal wfe.

Suppose E has a type assignment and E' is the result of a single evaluation step upon E (i.e., either a β or δ-reduction). Consider a type assignment to E' which gives each subexpression in E' the type of its father[+] in E. We shall show that such an assignment is valid in the sense that Rules 1 through 4 are obeyed.

It should be clear that if the evaluation step was a δ-reduction then the new type assignment is valid. For



because of our restriction on δ-reduction.

Theorem 1:

If E has a valid type assignment T, and E' results from E by a β-reduction, then E' has a valid type assignment T' in which every subexpression is assigned the same type as its father in E.

[+] Recall that father is a function associating subexpressions in E' with subexpressions in E. (See Chapter II, section C.)

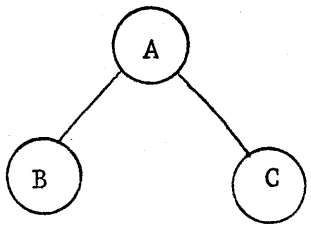
Proof: Assume that a redex of the form $(\lambda x.M)N$ in E is contracted, yielding E'.

We need only show that the type assignment, T', is a valid one, i.e., that it obeys Rules 1, 2, 3 and 4.

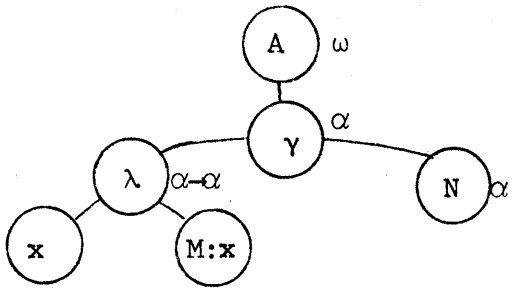
As for Rule 1: Suppose two occurrences of some variable, V, are free in a subexpression of E'. Consider the largest expression, S, in which they are both free; either S is E', the entire expression, or it serves as the body of a λ-expression having V as its bound variable. In either case the fathers of the two occurrences must have been free in the father of S. This is a consequence of the fact that a free variable cannot be captured by a β-reduction. Since T was a valid type assignment, the fathers of these occurrences of V had the same type. Therefore, they are assigned the same type by T'.

The validity of T' with respect to Rules 2 and 3 depends upon their being obeyed for every three nodes, A, B and C, which occur in E' and have the configuration shown below.

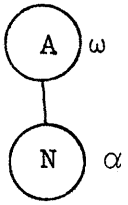
If the fathers of A, B and C were in the same relation to each other in E, then Rule 2 or 3 (whichever applies) must be obeyed by the validity of T. Thus, we need only consider those cases where their fathers were not adjacent in the same way. The only such cases are when B or C is the son of the rand of the redex contracted or the son of the body of the rator.



First, let us dispense with the special case where $M \equiv x$; i.e., the redex has the form $(\lambda x.x)N$. Schematically, we must have



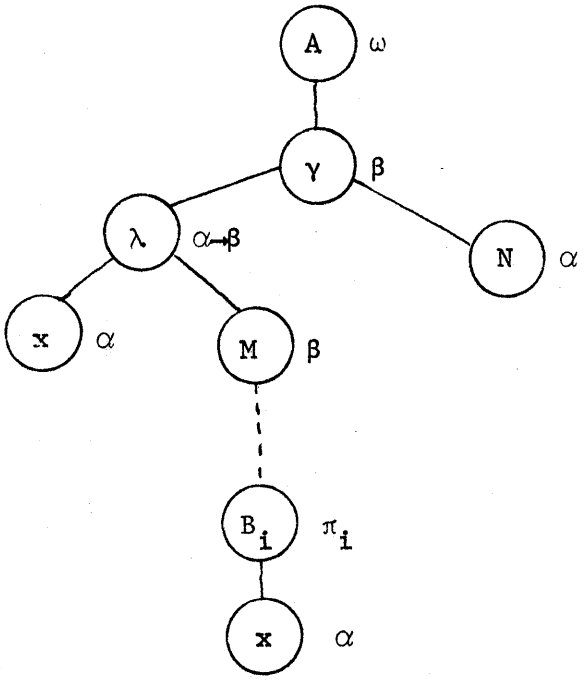
in E with the type assignment shown; and



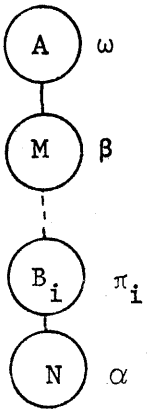
in E' with each node assigned the type of its father in E.

Now if $M \not\equiv x$, let $B_1,\ldots,B_N$ $(N \geq 0)$ be the nodes in M immediately superior to the free occurrences of x in M.

Suppose the relevant nodes are assigned types as shown below.



Contraction of the redex yields



Here each node has been assigned the type of its father. Since the nodes immediately inferior to A and the B_i in E have the same types as the nodes inferior to A and the B_i in E, the assignment must continue to be valid with respect to Rules 2 and 3.

Since β-reduction does not introduce any new constants every constant in E' is the son of one in E and thus has its proper type. Hence, Rule 4 is obeyed.

QED

This theorem and the preceding consideration show that the type system is adequate in the sense that it effectively restricts the class of wfes to those which will not lead to don't care situations during evaluation. The system requires that these don't care conditions be codified as permanent types for all the constants, and that the reduction rules for constants be compatible with their types. The substance of the theorem is that β-reduction is compatible with the type rules for all wfes.

It is to be noted that the adequacy of the type system does not depend upon the order of reduction employed by the evaluator (left to right). There are many expressions which cannot have a type assignment which the evaluator will process without ERROR. For example, $(\lambda x.5)(3\ 1)$ has no type assignment but yields 5 upon left-to-right evaluation. In general, a wfe is legal (has a type assignment) only if every subexpression is legal.

E. An Algorithm for Type Assignments

We claim that the type checking process carried out by a compiler for a language such as ALGOL-60 is analogous to deriving a type assignment for a wfe. We develop the analogy as follows:

(a) We assume that all the primitive operators like +, -, etc. are constants with functional types known a priori, to the compiler.

(b) The declarations (e.g., real x or real procedure f) serve to assign types to various nodes in the tree which are variables.

(c) The compiler endeavors to extend the partial type assignment to a complete one and rejects the program if it cannot.

For our simple language, explicit type declarations are unnecessary if the only purpose of a type assignment is to insure that no ERRORs occur. All that is needed is a check to insure that a given wfe has at least one type assignment, and we shall show how this can be done fairly efficiently. Of course, practical languages use type declarations for other things such as determining storage allocation, but we are not concerned with these problems here.

The following algorithm checks a wfe for potential type ERRORs by attempting to give it a type assignment. We shall intersperse an example with the statement of the algorithm.

Step 1.

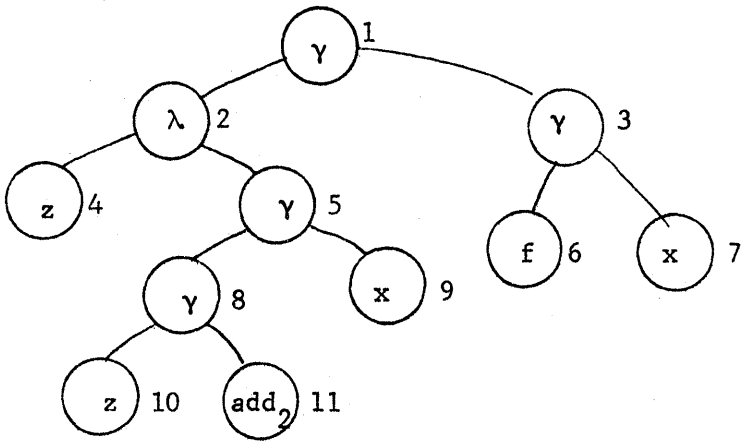
Assign a distinct type variable, V_i, to each node of the wfe and write down the equations required by Rules (1) - (4) in the definition of type assignment.

Example 4.

Consider the wfe

$$(\lambda z.z\ add_2\ x)(fx)$$

We shall use integers instead of V_i's as type variables and assign them as shown below.



The equations required by (1) are

$$7 \equiv 9$$

$$4 \equiv 10$$

The equation required by (2) is

$$2 \equiv 4 \rightarrow 5$$

The equations required by (3) are

$$10 \equiv 11 \rightarrow 8$$

$8 \equiv 9 \rightarrow 5$

$6 \equiv 7 \rightarrow 3$

$2 \equiv 3 \rightarrow 1$

The equation required by (4) is

$11 \equiv N \rightarrow N$

Now the problem is simply to discover if there is a "solution" to these equations; i.e., whether there are type expressions $T_1,\ldots,T_{11}$ for which the equations are true. A simple iterative process will suffice.

Step 2.

Use any equivalences of the form $V_i \equiv V_j$ or $V_i \equiv t_i$ (i.e., t_i is an atomic type) to eliminate V_i from the equations of the form $A \equiv B \rightarrow C$.

If any contradictions appear, halt; there is a type error. A contraction is any equation of the form $t_i \equiv t_j$ for $i \neq j$ or $t_i \equiv A \rightarrow B$ for any A, B.

Example 4. (continued)

We eliminate 7 and 4 from the equations and rewrite

a. $2 \equiv \underline{10} \rightarrow 5$

b. $10 \equiv 11 \rightarrow 8$

c. $8 \equiv 9 \rightarrow 5$

d. $6 \equiv \underline{9} \rightarrow 3$

e. $2 \equiv 3 \rightarrow 1$

f. $11 \equiv N \rightarrow N$

Step 3.

If two equations have the same left-hand side, $A \equiv B \rightarrow C$ and $A \equiv B' \rightarrow C'$, eliminate one and add the equivalence

$$B' \equiv B$$

$$C' \equiv C$$

Return to Step 2.

Example 4. (continued)

We note that equations a. and e. fulfill the condition. Eliminate e. and add

$$3 \equiv 10$$

and $1 \equiv 5$

Carrying out Step 2, we have

a. $2 \equiv 10 \rightarrow 5$

b. $10 \equiv 11 \rightarrow 8$

c. $8 \equiv 9 \rightarrow 5$

d. $6 \equiv 9 \rightarrow 10$

f. $11 \equiv N \rightarrow N$

Evidently, no contradictions have appeared, and Step 3 is no longer applicable.

Step 4.

If the conditions of Step 3 are not met, check all the equations of the form $A \equiv B \rightarrow C$ for circularities. For example, the two equations

$$V_1 \equiv V_2 \rightarrow V_3$$

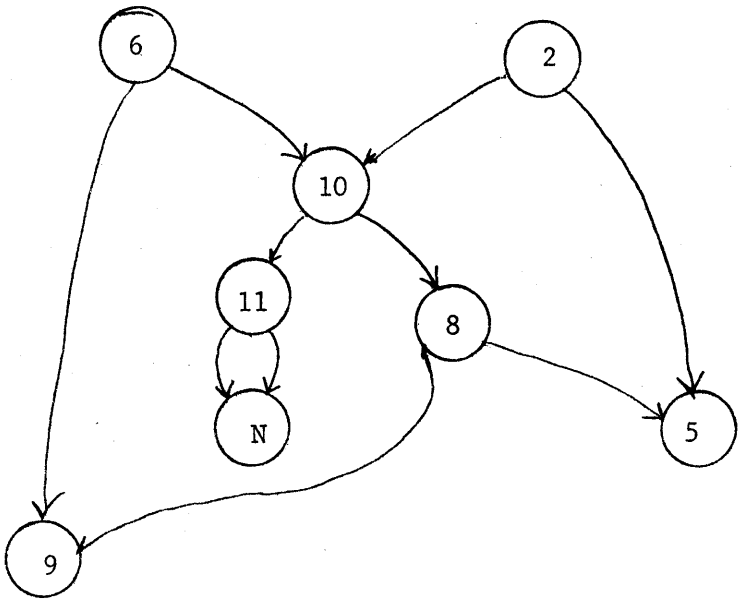
$$V_2 \equiv V_3 \rightarrow V_1$$

contain a circularity requiring that

$V_1 \equiv (V_3 \rightarrow V_1) \rightarrow V_3$; i.e., that V_1 be a proper subpart of itself. If a circularity exists, there can be no type assignment. Otherwise, there is at least one type assignment.

Example 4 (concluded).

By displaying equations a. - f. in graphical form, we see that they contain no circularity.



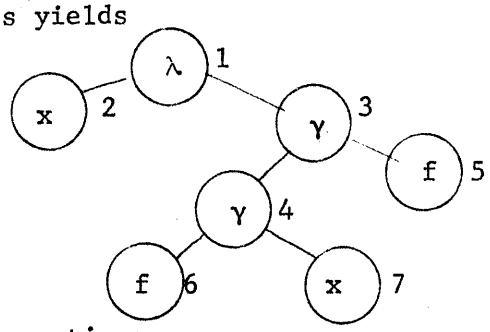
The type variables, 9 and 5, are unrestrained in that we may choose any type expressions for them to define a type assignment. For example, by choosing $9 \equiv 5 \equiv N$, we define a type assignment in which $8 \equiv N \to N$, $10 \equiv (N \to N) \to (N \to N)$, etc.

This completes the statement of the algorithm.

Example 5. Consider the wfe

$$(\lambda x.fxf)$$

Assigning type variables yields



and (among others) the equations

$$6 \equiv 5$$

$$6 \equiv 7 \to 4$$

$$4 \equiv 5 \to 3$$

which lead to a circularity

$$6 \equiv 7 \to (6 \to 3).$$

F. Drawbacks of the Type System

There are many programmers who feel that the type systems of higher level languages, with their attendent requirement that all variables be declared, are a nuisance. Sooner or later, most FORTRAN programmers feel the need to "fool" the compiler through the use of EQUIVALENCE statements.+ The use of such devices leaves one open to such epithets as "bit-twiddler" or "programming bum."

We shall endeavor to show that the type system we have imposed on our simple language has a genuinely deleterious effect on its expressive power. While the implications of this for practical programming languages are not direct, we claim that certain of their type declaration requirements in the area of function variables are closely related to the type system presented here.++

An attractive feature of the type-free λ-calculus is that one may relentlessly abbreviate a given wfe by abstracting on common subexpressions. In doing so, we may pass from a wfe with a type assignment to one with none.

Example 6. Suppose a particular language has two atomic types, N (for number) and S (for character string). Further, suppose square is a

---

+ A FORTRAN EQUIVALENCE statement allows one to assign two distinct variables to the same storage location. By letting one variable have type integer and the other have type real, one can treat the contents of the location as either kind of number.

++ Significantly, ALGOL-60 does not require full declarations of function variables; but two recent languages, ALGOL-68 [18] and BASEL [19] do require that every function designator (i.e., variable) have its domain and range types declared.

function of type N → N and trunc is a function of type S → S. Then the wfe

(... square (square 5) ... trunc (trunc "ABC") ...)

may be abbreviated to

(λ twice .(...twice square 5 ... twice trunc "ABC" ...))

(λf.λx.f(f x))

Now the second expression cannot have a type assignment because twice appears in contexts where it must have both of the types

(N → N) → (N → N) and (S → S) → (S → S).

Example 7. The wfe Y ≡ λf.(λh.f(hh))(λh.f(hh)) obviously cannot have a type assignment as it contains a subexpression (hh). (See Example 2.) This is rather significant since Y is used to effect recursive definitions.

The most revealing fact about the effect of imposing a type system is given by the following:

Theorem 2.

If a wfe has a type assignment, then it has a β-normal form.

Proof: Assume a type assignment has been made to E. For any β-redex in E define the degree of that redex to be the number of " → "s in the type expression assigned to its rator.

If E contains a β-redex of degree m and no β-redex of greater

degree, contract a redex $(\lambda x.M)N$ with degree m such that N contains no other redex of degree m. Evidently, such a redex can always be found. (Choose the right-most redex of degree m, for example.)

Now we show that the resulting expression, E', must have one less redex of degree m than E does.

Any redex, $(\lambda y.K)L$ in E' must be the son of a combination in E. The rator of that combination is either

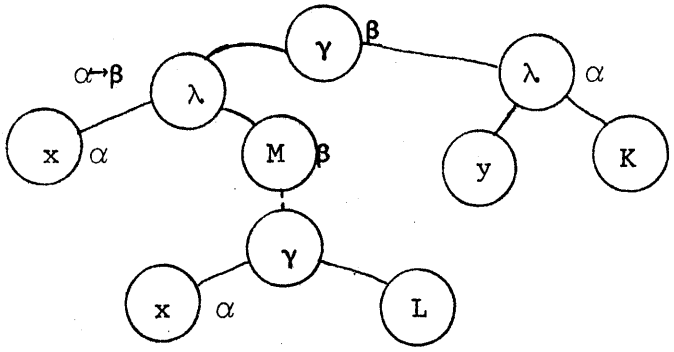
(1) a λ-expression

or (2) a variable

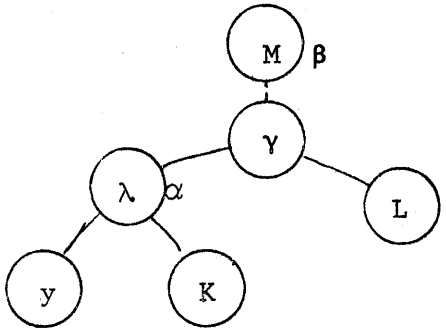
or (3) a combination

In case (1) the father itself is a redex. If it is a redex of degree less than m, its son also has degree less than m. If it has degree m, then it did not occur in N and therefore has only one son (namely, $(\lambda y.K)L$). Thus, since the redex contracted, $(\lambda x.M)N$, has no son, there is one less redex of degree m whose father was a redex.

In case (2) a λ-expression has replaced a variable in a combination. Since only free variables of M are replaced by the contraction, the redex must have had the following form:

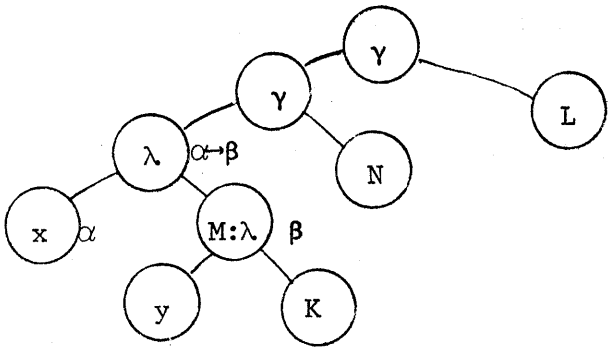


and its contraction is of the form

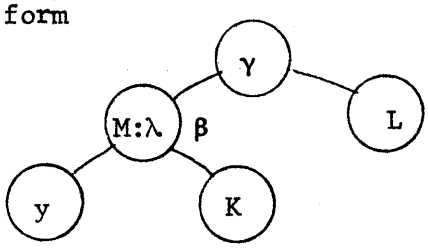


Now since α contains fewer " $\rightarrow$ "s than $\alpha \rightarrow \beta$, this redex in E' must have degree less than m.

In case (3) a λ-expression has replaced a combination. Since β-contraction replaces only one combination in an expression, namely, the redex itself, the redex $(\lambda x.M)N$ must be a rator of another combination. There are two cases to consider: First, M may be a λ-expression so its immediate context appears as follows:

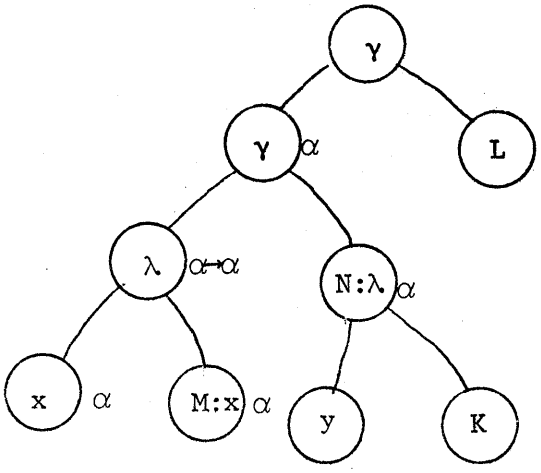


and its contractum is of the form

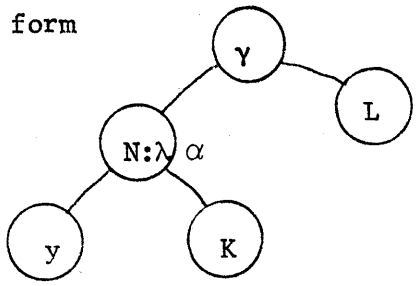


Since β contains fewer " $\rightarrow$ "s than $\alpha \rightarrow \beta$, this redex in E' must have degree less than m.

In the second sub-case of (3), M may be x, (i.e., $\lambda x.M \equiv \lambda x.x$) and $N \equiv (\lambda y.K)$. Schematically,



and its contractum has the form



Since α contains fewer " $\rightarrow$ "s than $\alpha \rightarrow \alpha$ the new redex must have degree less than m.

Thus, we can reduce any expression possessing a type assignment to β-normal form by choosing the redex according to the aforementioned rule at every stage. Notice that redexes with degree less than m may increase at any stage, but we will eventually reach a stage where m=1 and thus eliminate all the remaining redexes.

QED

Corollary 2. If a wfe has a type assignment, then all its sub-expressions have a β-normal form.

Proof. Immediate, since each subexpression must have a type assignment.

This result was proved by Curry in his theory of functionality. The proof here is quite different and was arrived at independently.

This theorem illustrates that imposing the type system on the λ-calculus restricts its power in a quite significant way. Not only do we eliminate the possibility of E-undefinedness through don't care situations, we also exclude non-terminating computations, or

ω-undefinedness, from the system. The colloquialism "throwing out the baby with the bath-water" seems appropriate here; our efforts to make the language safe have made it too weak to be universal.

G. An Extended System

In order to add perspective to the treatment of types presented in the foregoing, we briefly outline an extended language and type system.

The formal syntax is extended by adding several operators (see Chapter II, section A) to the system. In addition to λ, γ, the variables and the constants we have the following operators:

if with degree 4

rec with degree 1

, with degree 2

h with degree 1

t with degree 1

switch with degree 3

tagl with degree 1

tagr with degree 1

To specify the written representation of expressions involving these new operators, we give the BNF for a fully parenthesized wfe.

< wfe > ::=

(<wfe> <wfe>)	(combination)
(λ <variable> . <wfe>)	(λ-expression)
(if <wfe> = <wfe> then <wfe> else <wfe>)	(conditional)
(rec <wfe>)	(recursion)
(<wfe>, <wfe>)	(pair)
(h <wfe>)	(head selection)

(t < wfe >) | (tail section)

(switch < wfe > into < wfe > or < wfe >) | (switch)

(tagl < wfe >) | (tag left)

(tagr < wfe >) | (tag right)

< variable > | < constant >

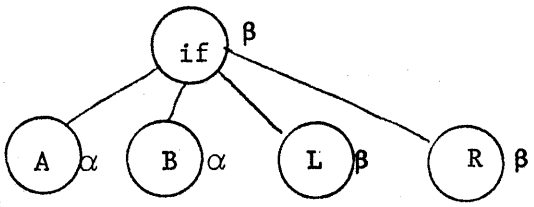
We shall present and discuss the type rules and reduction rules for the new constructs seperately.

1. Conditionals.

The type constraints on

(if A = B then L else R)

are depicted below



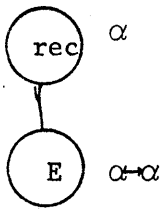
That is: A and B must have the same types while L, R, and the entire expression must have the same type.

A conditional expression is reducible if both A and B are in normal form with respect to all other reductions. If A and B are identical, then it is reducible to L, otherwise, it is reducible to R.

The rationale for the type rules is as follows: Requiring that the expressions A and B have the same type reflects a prejudice

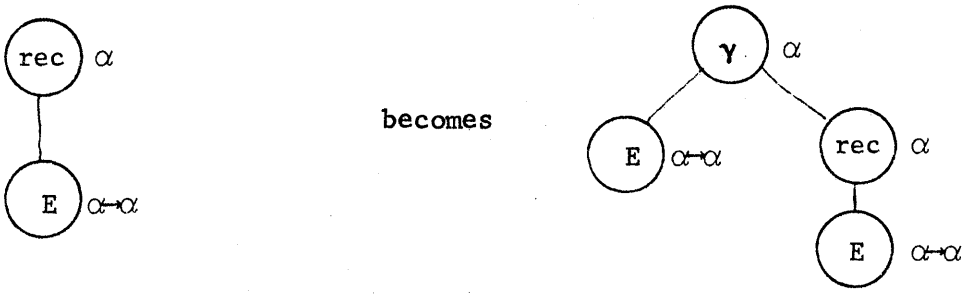
that only objects of the same sort can be compared. It also allows an implementation to choose the same representation for objects of different type and still be able to carry out the test. It is also reasonable to require α to be an atomic type; but we do not bother to do so. The requirement that L, R and the <u>if</u> node share a common type is simply to assure that the type assignment is preserved by a reduction.

2. <u>Recursion</u>. The type rule for (<u>rec</u> E) is shown below



That is, E must be a function type with identical domain and range, α, and the <u>rec</u> node must have type α.

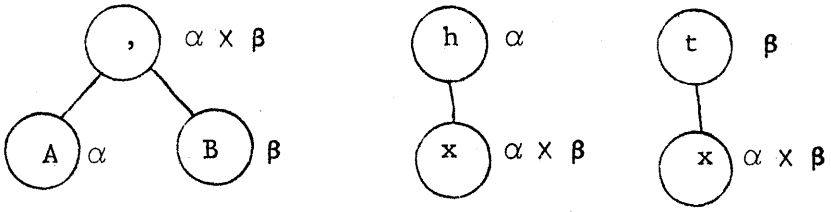
The reduction rule is: (<u>rec</u> E) may be replaced by (E (<u>rec</u> E)). Pictorially



Thus, <u>rec</u> E plays the same role as YE in the type-free λ-calculus. It introduces the possibility of a wfe with a type assignment but no normal form. Notice that the type assignment is preserved by the reduction rule.

3. Pairs.

We introduce a new kind of type expression, the cartesian product. If A and B are type expressions, then A $\times$ B is a type expression. This type corresponds to McCarthy's cartesian product in [17]. The type rules for pairs, head-selections and tail-selections are depicted below.



The reduction rules are

h(A, B) becomes A

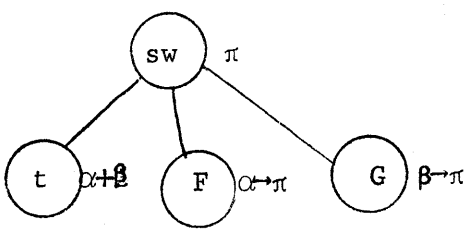
t(A, B) becomes B

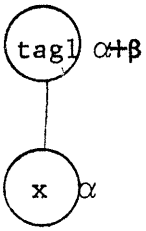
Hence a type assignment is preserved under these reduction rules.

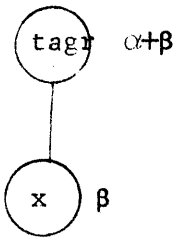
4. Switches and Tags

The mechanism presented here is intended to restore some of the freedom found under a type-free system. It allows one to declare a variable with an "ambiguous type" and test it dynamically. It is our version of McCarthy's disjoint union [17] and the union of ALGOL-68 [18] and BASEL [19].

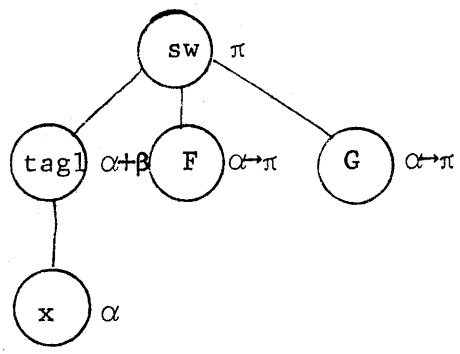
First, we introduce a new type. If A and B are type expressions, A + B, the join of A and B, is a type expression. The type rules governing switch statements and tag statements are depicted below.



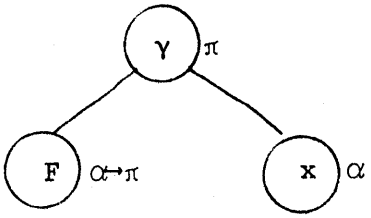




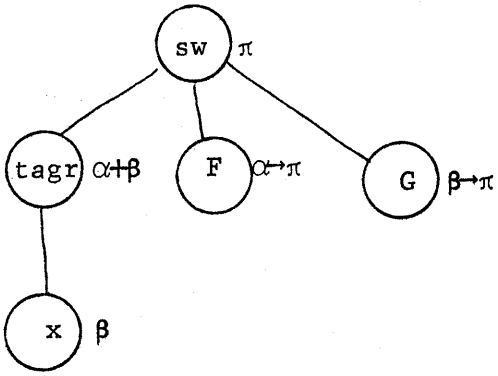
The reduction rules are



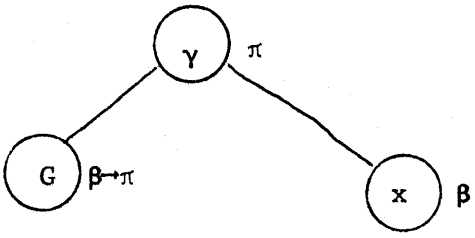
becomes



and



becomes



Intuitively, a switch operator takes a tagged object, x and two functions, F and G. It applies F to x if x was tagged with tag1. It applies G to x if it was tagged with tagr. Thus, it is possible for x to be of different types at different times.

For example, if x were of type integer and strings, the expression

switch x into (λy.y+1) or length

would have type integer (assuming length ϵ string $\rightarrow$ string).

5. Properties of the Extended Language

If we define the father function for each of the new kinds of reduction in an appropriate way it is easy to prove an extension to Theorem 1. Specifically, if a wfe in the extended language has a type assignment then any wfe that is reducible to has a type assignment in which each son has the type of its father.

It should also be clear that the type assignment algorithm of section E can be extended to account for the new types and type rules. (We continue to assume that type expressions are equal only if identical.)

Any wfe containing the rec operator obviously does not have a normal form since reduction of (rec E) creates a new occurrence of (rec E). Thus no extension of Theorem 2 holds. If a wfe contains no occurrence of rec and has a type assignment, we believe that it will have a normal form; but we are not able to prove this at present.+

+ The argument of Theorem 2 cannot be extended to wfes containing conditionals since types of redexes inside of the test part at a conditional are effectively de-coupled from the type of the conditional expression itself.

H. Summary

a. The question of whether a given wfe will cause an ERROR when evaluated is undecidable.

b. Imposition of a "reasonable" type declaration system on the language defines a decidable subset of wfe's which do not cause ERRORs.

c. Explicit declarations are not required to carry out the type checking algorithm.

d. The subset does not contain many useful wfes that would not cause ERRORs, most notably Y or any other wfe not having a normal form.

e. Some of the power of the language can be recovered while retaining the type system by introducing the special operator, rec.

CHAPTER V

Conclusions

A. Support for the Thesis

Our contention that recursion and type declarations can be given reasonable semantic interpretations is vague but not vacuous. To be sure, the reasonableness and adequacy of a semantic model are always debatable; but we are happy to defend the interpretations presented in the foregoing as being reasonable and fairly successful in reflecting the mechanisms in question.

We claim that the notion of a fixed-point of a functional is natural and that an operation on functionals which produces fixed-points is conceivable, if not obviously computable. The main contribution of Chapter III is to establish a semantic characterization of the fixed-point produced by the operation corresponding to Y, an obviously computable operation. Specifically, we introduce the relation of extension ($\supset$) between wfes and show how it is related to the simpler idea of one function containing another (Theorem 3, p. 50). We then show that YF is the minimal or "worst" fixed-point of F in the sense that any other fixed-point of is an extension of it (Corollary 7(a), p. 68). The apparent lack of a natural set-theoretic model for the λ-calculus prevents us from presenting a more precise interpretation of YF.

The relevant (to our thesis) postulates of the type system presented in Chapter IV are:

1. Types are sets of values.
2. A declaration is a statement that a particular variable (more generally, an expression) assumes values only from a particular type.
3. The universe of values is partitioned into atomic types, function types, functional types, etc.; i.e. each value belongs to one and only one type. (The Stratification Hypothesis)

Postulates 1 and 2 are neither original nor controversial. We claim that they constitute a quite reasonable interpretation of type declarations as found in extant programming languages. Postulate 3 is implicit in any language which calls for declaration of the domain and range of function variables. It appears to be the simplest generalization of the assumption that an object cannot be both a function and an atom.

Therefore, we construe the initial part of Chapter III (sections A through G) as a rational construction of a language with type declarations from the type-free λ-calculus and the aforementioned postulates. This construction is shown to be lacking in section F, especially by Theorem 2 (p.107), which shows that the type system makes the λ-calculus an uninteresting programming language; i.e. one without non-terminating computations.

Rather than reject the postulates we adopt the simple expedient of introducing recursion by fiat, with the new operator rec (p.114). This allows the possibility of non-terminating computations. (The other

extensions are made for less vital reasons, mainly to add perspective.) This extension of the operators is not entirely without semantics. Indeed, the results of Chapter III provide a natural way of interpreting (rec F) - precisely the way we interpret (Y F). The only distinction is that rec denotes no object in a universe of discourse while Y might, if we could think of a suitable universe. The expression (rec F), however, can be interpreted in the stratified universe: if $F \in \alpha \to \alpha$ then (rec F) $\in \alpha$ and is the minimal fixed point of F.

To conclude: we assent that the analyses of recursion and type declarations develop interpretations that are:

1. Semantic in the sense that they relate expressions to objects in a reasonable universe.
2. Adequate in that they reflect the purely mechanical definitions of the notions.

B. Directions for Future Research

We confess that the work of Chapter IV is only partially directed at the semantic analysis discussed above. We are interested in more concrete problems associated with static type checking. Our work has suggested a few alternate possibilities for the design of type declaration systems, and we outline them here. Each involves rejections or modifications of the three postulates presented in section A above.

1. Overlapping Types

Requiring each object in the universe of values to belong to only one type seems arbitrary; this has been pointed out by Landin (in personal communication). We relax postulate 3 as follows: The

universe is still stratified in the sense that atoms, functions, etc. are distinct, but the types at any level may intersect. For example, certain atomic types such as real and integer may share members (e.g. 3) and certain functional types such as integer → integer and real → real may share members (e.g. the function abs, for absolute value).

We may now consider additional operations on types such as union ($\cup$) and intersection ($\cap$). Declaring that variable x ϵ real $\cup$ string would mean that the value of x would either be a real or a string at any time. Declaring that a variable abs ϵ integer → integer $\cap$ real → real would mean that abs was a function which maps integers into integers and reals into reals.

The rules for type assignments (p.93) can be relaxed. Specifically we might restate rule 3 to require only that certain inclusion relations obtain between the function, argument, and result of a combination.

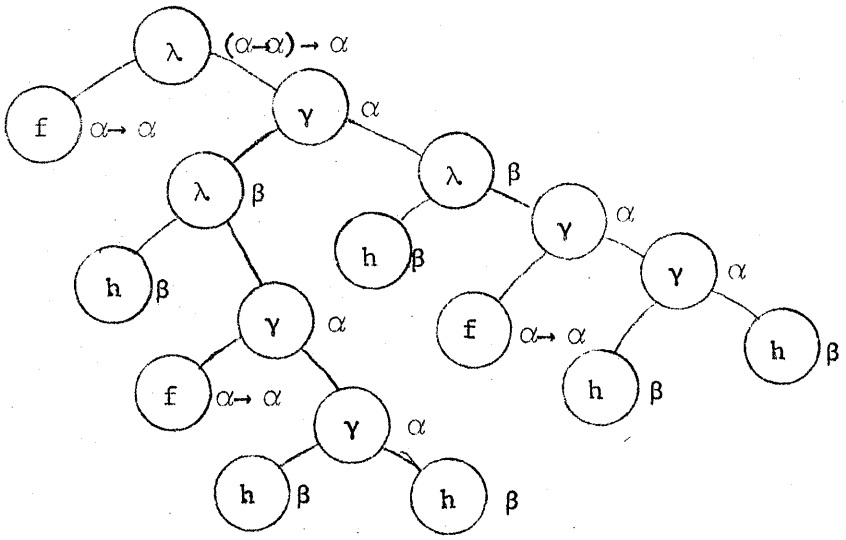
The usefulness of such semantic revisions of the type system depends on our ability to devise the appropriate type-checking algorithm to match the semantics. There seems little point in devising a declaration language which cannot be processed by a type-checker.

2. Circular Types

We see no pragmatic drawbacks in permitting the circular type expressions which were explicitly disallowed by Step 4 of the type assignment algorithm (p.100). Examination of the proof of Theorem 1 (p.96) reveals that it was unnecessary to assume that the type expressions used in the type assignment were finite, (i.e., non-circular). Thus, it appears that a wfe with a type assignment involving circular type expressions will not cause errors.

We have only limited intuition about what kind of an object a circular type expression might denote (possibly, a set of functions, some of which have themselves as arguments and values). Circular types are difficult to reconcile with postulates 1 and 3, but squeamishness on semantic grounds seems counter-productive here.

If we permit circular type expressions, Y can be given the type assignment shown below



where $\beta = \beta \rightarrow \alpha$. It is interesting to note that the top-most node does not have a circular type since it involves only α.

The usefulness of circular types is more obvious if we consider cartesian products and joins as defined in the extended language (p.). We need the circular type

$$S = A + S \times S$$

to describe LISP S-expressions (which seem to have become the sine qua non for type description systems). We simulate S-expressions as follows:

write	tagr(x,y)	for	cons [x;y]
write	h x	for	car [x]
write	t x	for	cdr [x]

To create an "atom" from any object, x, write tagl x.

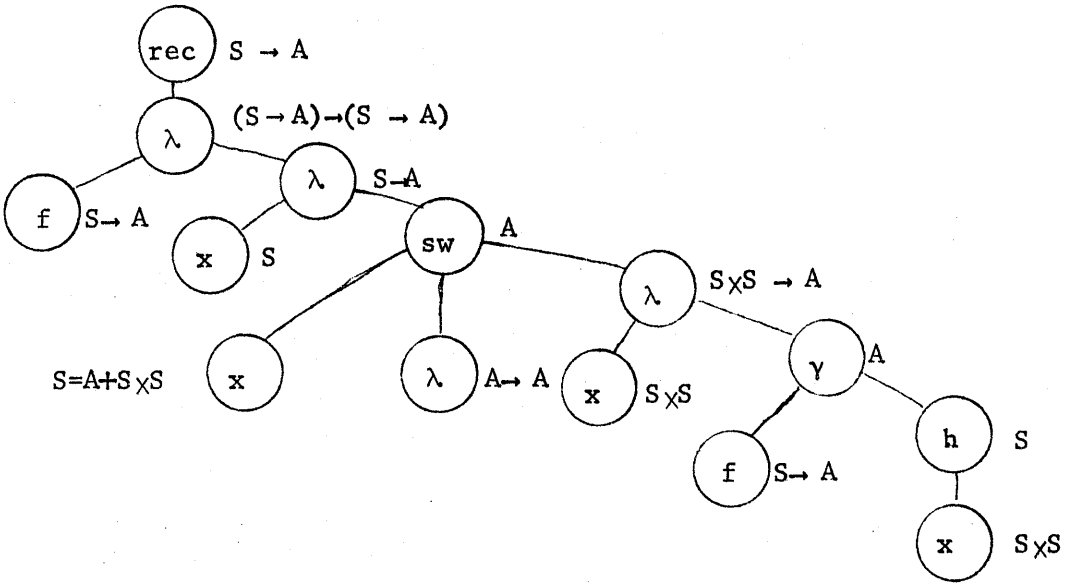
The LISP function definition

ff[x] = atom [x] → x; T → ff [car x]

can then be rendered as

ff = rec (λf. λx. switch x into (λx.x) or (λx. f(hx)))

We display the corresponding type assignment below.



Allowing circular functional types does not solve all problems. For example,

(λx.xx) twice square 3

reduces to

square^4 3

but the revised type assignment algorithm will still reject it, reaching a contradiction of the form N = N → N.

3. Programmers' Polymorphic Functions

This term was coined by Christopher Strachey [26] to describe functions such as twice of Example 6. The problem is: design a language and type system in which a programmer may define functions whose parameters may have different types for different calls of the function.

To do so we must reject postulate 2.

A simple approach to a solution of this problem is based upon the observation that a wfe which cannot be given a type assignment is sometimes reducible to one that can.

A type checking procedure which would admit certain polymorphic function definitions would be the following:

1. Perform a complete reduction relative to all the β-redexes in the wfe.
2. Apply the type assignment algorithm of Section E to the resulting wfe.

For example, consider the wfe

$(\lambda twice.(twice\ square\ 3),(twice\ trunc\ "ABC"))(\lambda f.\lambda x.f(f\ x))$

A single reduction yields

$(\lambda f.\lambda x.f(f\ x))\ square\ 3,\ (\lambda f.\lambda x.f(f\ x))\ trunc\ "ABC"$

which may be given the type assignment $N \times S$.

What can we say about the class of wfes that would be accepted by this revised type-checking algorithm? (E.g. does theorem 1 of Chapter IV still hold?) How can the algorithm be made more efficient? We leave these questions for future investigation.

4. Completely Dynamic Types

The most radical approach consists in rejecting the notion of static type checking entirely.

A persuasive case can be made for languages which do not attempt to constrain the values of variables in any way. Three such languages are LISP, PAL [20], and BCPL [21]. To be sure, each of these

languages has distinctly different types of objects in its universe of values as evidenced by the explicitly specified undefinedness of certain primitive operations (e.g., car applied to an atom in LISP). We do not argue that the programmer should not know what kinds of values the variables in his program will assume, but we cannot think of a formal type declaration system adequate to describe many of the possibilities.

One remedy for this problem is the development of more elaborate and expressive type description facilities. The work of Perlis and Galler [22], Standish [23], van Wijngaarden [18], and Fischer and Jorrand [19] include such developments. While the systems presented incorporate many interesting ideas, none allows the freedom of a language without declarations.

How does one implement a language without type declarations? We cannot think of any entirely satisfactory answer to this question. Two extreme alternatives are

a. Write an interpreter for the language which detects type errors and reports on them when they occur (PAL).
b. Write a compiler and allow type errors to cause implementation-dependent responses (BCPL).

The first alternative makes an implementation inefficient while the second makes programs difficult to debug and the transfer of programs between machines a dubious venture.

Finding a reasonable distribution of type-checking responsibility between the programmer, the compiler and the run-time system calls for creative language design. We consider this a difficult but important area for further research.

C. Remarks

The ideas sketched above illustrate the advantages and shortcomings of semantic analysis as a tool for language design. The possibility of overlapping types is clearly suggested by the semantics developed for type declarations. By thinking of types as sets we open the way for many extensions of the type system which would not occur to us were we to restrict ourselves to pure pragmatic reasoning. The problem remains, however, of mechanizing the set-theoretic extensions--a highly non-trivial problem. In contrast, the suggestions for circular types and polymorphic functions are almost entirely derived from our understanding of the mechanics of the type system. We have no obvious semantic basis to guide their development, but we have few doubts about our ability to implement them as described. If these ideas contribute to a useful type declaration system it is folly to ignore them simply because we cannot concoct semantics to explain them. Such a policy would be like that of a caveman who avoids the use of fire until a suitable fire-god is identified or a baseball pitcher who eschews curve-balls until the aerodynamic principles are established.

We believe that the art of programming language design is forwarded by both semantic analysis and pragmatic innovation. Our dissertation follows the former approach, which heretofor seems to have been neglected in the pursuit of bigger and better languages. In a sense, we have been led a merry chase by the ad hoc inventors of languages; but we have enjoyed the exercise!

REFERENCES

[1] Church, A., The Calculi of Lambda Conversion, Annals of Mathematics Studies, No. 6, Princeton, N.J., Princeton University Press, 1941.

[2] Perlis, A. J., General Discussion at Close of a Working Conference on Mechanical Language Structures, CACM, Vol. 7, No. 2, February, 1964, p. 136.

[3] Quine, W. V. O., Word and Object, The M.I.T. Press, Cambridge, Mass., 1960.

[4] Landin, P. J., "A Correspondence between ALGOL-60 and Church's λ-notation," CACM, Vol. 8, February and March, 1965, pp. 89-101, 158-165.

[5] Landin, P. J., "The Next 700 Programming Languages," CACM, Vol. 9, No. 3, March, 1966, pp. 157-164.

[6] Morris, C. W., "Foundations of the Theory of Signs," International Encyclopedia of Unified Science, Vol. 1, No. 2, University of Chicago Press, Chicago, 1955.

[7] Koestler, A., The Ghost in the Machine, MacMillan, New York, 1968.

[8] Whorf, B. L., Language, Thought and Realty: Selected Writings of Benjamin Lee Whorf, (J. B. Carrol, ed.), The M.I.T. Press, Cambridge, Mass., 1956.

[9] McCarthy, J., et al., LISP 1.5 Programmer's Manual, The M.I.T. Press, Cambridge, Mass., 1962.

[10] Newell, A., ed. Information Processing V Manual, Prentice Hall, Englewood Cliffs, N.J., 1961.

[11] Curry, H. B. and Feys, R., Combinatory Logic, North Holland, Amsterdam, 1958.

[12] Rogers, H., Jr., Theory of Recursive Functions and Effective Computability, McGraw-Hill, New York, 1967.

[13] Landin, P. J., "A λ-calculus Approach," in Advances in Programming and Non-numerical Computation, Permagon Press, New York, 1966.

[14] Scott, D., "A System of Functional Abstraction," Unpublished Notes, Stanford University, September, 1963.

[15] Bohm, "The CUCH as a Formal and Description Language," in Formal Language Description Languages for Computer Programming, T. B. Steel, Ed., North Holland, Amsterdam, 1966.

[16] Kleene, S. C., Introduction to Metamathematics, D. Van Nostrand, Princeton, N.J., 1950.

[17] McCarthy, J., "A Basis for a Mathematical Theory of Computation," in Computer Programming and Formal Systems (P. Braffort and D. Hirschberg, eds.), North Holland, Amsterdam, 1967.

[18] Van Wijngaarden, et al., Draft Report on the Algorithmic Language ALGOL 68, Mathematish Centrum, Amsterdam, 1968

[19] Fischer, A. E. and Jorrand, P., "Basel: The Base Language for an Extensible Language Facility," Computer Associates, Inc., Report CA-6806-2811, Wakefield, Mass., 1968.

[20] Evans, A., "PAL - A Reference Manual and a Primer," M.I.T., Department of Electrical Engineering, Cambridge, September, 1968.

[21] Richards, M., "BCPL Reference Manual," Project MAC Memorandum M-352-1, February, 1968.

[22] Galler, B. A.and Perlis, A. J., "A Proposal for Definitions in ALGOL," CACM, Vol. 10, No. 4, April, 1967, pp. 204-219.

[23] Standish, T. A., "A Data Definition Facility for Programming Languages," PhD. Dissertation, Carnegie Institute of Technology, Pittsburgh, Penna., 1967.

[24] Bohm, C., "Alcune Proprieta Delle Forme β-η-normali nel λ-K-calcolo," Consiglio Nazionale Delle Ricerche, No. 696, Rome, 1968 (Italian).

[25] Wozencraft, J. M., "Programming Linguistics," Classroom Notes (6.231), M.I.T., Department of Electrical Engineering, Cambridge, Mass., 1968.

[26] Strachey, C,,"Fundamental Concepts in Programming Languages", NATO Conference, Copenhagen, 1967.

Biographical Note

James H. Morris, Jr. was born in Pittsburgh, Pennsylvania, on October 11, 1941. He attended Shady Side Academy there, graduating in 1959. He received a B.S. in mathematics from Carnegie Institute of Technology where he was the recipient of a George Westinghouse Scholarship.

At M.I.T. he held a National Defense Education Act Fellowship for three years and a research assistantship at Project MAC in the following years. He received a master's degree in management from the Sloan School of Management in 1966.

In September, 1967, he married the former Susan A. Schumacher of Pittsburgh.

Mr. Morris is a co-author of On-Line Computation and Simulation: The OPS-3 System, published by the M.I.T. Press.

He is presently an assistant professor of Computer Science at the University of California at Berkeley.